

Model-Driven Engineering
for Distributed Real-Time Systems

Model-Driven Engineering for Distributed Real-Time Systems

*MARTE Modeling, Model Transformations
and their Usages*

Edited by
Jean-Philippe Babau
Mireille Blay-Fornarino
Joël Champeau
Sylvain Robert
Antonio Sabetta

ISTE

 **WILEY**

First published 2010 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms and licenses issued by the CLA. Enquiries concerning reproduction outside these terms should be sent to the publishers at the undermentioned address:

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

© ISTE Ltd 2010

The rights of Jean-Philippe Babau, Mireille Blay-Fornarino, Joël Champeau, Sylvain Robert and Antonio Sabetta to be identified as the authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

Library of Congress Cataloging-in-Publication Data

Model-driven engineering for distributed real-time systems : MARTE modeling, model transformations, and their usages / edited by Jean-Philippe Babau ... [et al.].

p. cm.

Includes bibliographical references and index.

ISBN 978-1-84821-115-5

1. Model-driven software architecture. 2. Electronic data processing--Distributed processing. 3. Real-time data processing. 4. UML (Computer science). I. Babau, Jean-Philippe.

QA76.76.D47M622 2010

005.2'732--dc22

2010027955

British Library Cataloguing-in-Publication Data

A CIP record for this book is available from the British Library

ISBN 978-1-84821-115-5

Printed and bound in Great Britain by CPI Antony Rowe, Chippenham and Eastbourne.



Table of Contents

Chapter Summary	xi
Chapter 1. Model Transformation: A Survey of the State of the Art.	1
Tom MENS	
1.1. Model-driven engineering.	1
1.2. Model transformation	2
1.2.1. Definitions.	2
1.2.2. Taxonomy	4
1.3. Model transformation languages.	5
1.4. Model transformation activities	8
1.5. Conclusion	14
1.6. Acknowledgements	14
1.7. Bibliography	15
Chapter 2. Model-Based Code Generation	21
Chris RAISTRICK	
2.1. Introduction	21
2.2. The model-driven architecture (MDA) process . .	22
2.3. The automated approach to code generation. . .	23
2.4. Domain modeling	25
2.5. The executable UML (xUML) formalism	29
2.6. System generation.	31

2.7. Executable UML to code mappings	34
2.8. Conclusions	41
2.9. Bibliography.	42

**Chapter 3. Testing Model Transformations:
A Case for Test Generation from Input Domain
Models**

43

Benoit BAUDRY

3.1. Introduction	43
3.2. Challenges for testing systems with large input domains	46
3.2.1. Large set of input data	46
3.2.2. Configurable systems	48
3.2.3. Grammarware and model transformations . .	48
3.2.4. Testing challenges	52
3.3. Selecting test data in large domains	52
3.3.1. Category partition	52
3.3.2. Combinatorial interaction testing	55
3.4. Metamodel-based test input generation.	58
3.4.1. Metamodel coverage criteria	59
3.4.2. Model and object fragments for test adequacy criteria	61
3.4.3. Discussion.	64
3.4.4. Automatic synthesis of test models	65
3.5. Conclusion	67
3.6. Acknowledgements	68
3.7. Bibliography.	68

**Chapter 4. Symbolic Execution-Based
Techniques for Conformance Testing**

73

Christophe GASTON, Pascale LE GALL, Nicolas RAPIN and Assia
TOUIL

4.1. Context	73
4.1.1. Conformance testing: an introduction	73
4.1.2. Conformance relation	74
4.1.3. An overview of the approach	78

4.2. Input output symbolic transition systems	79
4.2.1. Data types	79
4.2.2. Input/output symbolic transition systems . . .	80
4.2.3. Semantics	82
4.3. Symbolic execution	84
4.4. Conformance testing for IOSTS	87
4.4.1. Test purposes.	88
4.4.2. Preliminary definitions and informal description	89
4.4.3. Inference rules.	94
4.5. Concluding remarks	96
4.5.1. Choosing test purposes	96
4.5.2. Implementation issues	101
4.6. Bibliography	101

Chapter 5. Using MARTE and SysML for Modeling Real-Time Embedded Systems 105

Huascar ESPINOZA, Daniela CANCELA,
Sébastien GÉRARD and Bran SELIC

5.1. Introduction	105
5.2. Background	108
5.2.1. UML profiling capabilities.	108
5.2.2. SysML and MARTE modeling capabilities . .	111
5.3. Scenarios of combined usage.	113
5.3.1. Defining architecture frameworks	114
5.3.2. Requirements engineering.	115
5.3.3. System-level design integration	117
5.3.4. Engineering/quantitative analysis	120
5.4. Combination Strategies	125
5.4.1. Issues	125
5.4.2. Strategies	128
5.5. Related work.	130
5.6. Conclusion	133
5.7. Acknowledgements	134
5.8. Bibliography	134

Chapter 6. Software Model-based Performance**Analysis 139**

Dorina C. PETRIU

6.1. Introduction	139
6.2. Performance models	142
6.2.1. Queuing network models	144
6.2.2. Layered queuing network model	146
6.3. Software model with performance annotations. .	148
6.3.1. Performance domain model.	148
6.3.2. Source model example	152
6.4. Mapping from software to performance model . .	155
6.5. Using a pivot language: Core Scenario	
Model (CSM)	158
6.6. Case study performance model.	160
6.7. Conclusions	162
6.8. Acknowledgements	163
6.9. Bibliography.	163

**Chapter 7. Model Integration for Formal
Qualification of Timing-Aware Software Data
Acquisition Components 167**
Jean-Philippe BABAU, Philippe DHAUSSY and
Pierre-Yves PILLAIN

7.1. Introduction	167
7.2. System modeling.	170
7.2.1. Acquisition system modeling.	170
7.2.2. Case study	172
7.2.3. Formal modeling techniques	174
7.3. Variation points modeling	182
7.3.1. Variation points definition	184
7.3.2. CDL implementation	187
7.4. Experiments and results	189
7.4.1. Tools.	189
7.4.2. Experimentations	191

7.5. Conclusion	194
7.6. Bibliography	195

Chapter 8. SoC/SoPC Development using MDD and MARTE Profile 201

Denis AULAGNIER, Ali KOUDRI, Stéphane LECOMTE, Philippe
SOULARD, Joël CHAMPEAU, Jorgiano VIDAL,
Gilles PERROUIN and Pierre LERAY

8.1. Introduction	201
8.2. Related works	203
8.3. MOPCOM process and models	206
8.4. Application	210
8.5. System analysis	211
8.5.1. Requirement analysis	211
8.5.2. Functional analysis	212
8.5.3. Action language	213
8.6. Abstract modeling level	214
8.7. Execution modeling level	216
8.7.1. The platform independent model/application model in EML.	217
8.7.2. The platform model in EML	217
8.7.3. The platform specific model/allocation model in EML	218
8.7.4. Analysis model.	219
8.8. Detailed modeling level	220
8.8.1. Platform model	221
8.8.2. Allocation model.	222
8.9. Tooling Support	223
8.9.1. Process validation through metamodeling with Kermeta	223
8.9.2. Model transformation and generation with MDWorkbench platform	224
8.10. HDL Code Generation	225
8.10.1. VHDL code generation	226
8.10.2. Rhapsody integration	227
8.11. Conclusion	228

x Model-Driven Engineering

8.12. Acknowledgements	229
8.13. Bibliography	229
List of Authors	233
Index	237

Chapter Summary

Chapter 1

Model-driven engineering (MDE) is an approach to software development where the primary focus is on models, as opposed to source code. The use of models opens up new possibilities for creating, analyzing, manipulating and formally reasoning about systems at a high level of abstraction.

To reap all the benefits of MDE, it is essential to install a model transformation mechanism, that enables a wide range of different automated activities such as translation of models (expressed in different modeling languages), generating code from models, model synthesis, model improvement, model verification and model simulation. To achieve this, languages, formalisms, techniques, processes, tools and standards that support model transformation are needed. This chapter surveys the state of the art of model transformation, and discusses how it can be used to support some essential activities in MDE.

Chapter 2

This chapter explains how the combination of the OMG's Model-Driven architecture (MDA) process and the executable

UML formalism can be used to specify and build embedded software systems. It will deal specifically with:

- the Model-Driven Architecture principle of partitioning a system into domains for which we construct Platform Independent Models (PIMs);
- the use of Executable UML (xUML) for the construction of precise, complete PIMs that can be demonstrated and verified prior to implementation;
- automatic translation of the PIMs into Platform Specific Models (PSMs) and then into performance compliant code running on an embedded target.

Chapter 3

Model transformations can automate critical tasks in model-driven development. Thorough validation techniques are required to ensure their correctness. In this chapter we focus on testing model transformations. In particular, we present an approach for the systematic selection of input test data. This approach is based on a key characteristic of model transformations: their input domain is formally captured in a metamodel. A major challenge for test generation is that metamodels usually model an infinite set of possible input models for the transformation.

We start with a general motivation of the need for specific test selection techniques in the presence of very large and possibly infinite input domains. We also present two existing black-box strategies to systematically select test data: category-partition and combinatorial interaction testing. Then, we detail specific criteria based on metamodel coverage to select data for model transformation testing. We introduce object and model fragments to capture specific structural constraints that should be satisfied by input test data. These fragments are the basis for the definition of

coverage criteria and for the automatic generation of test data. They also serve to drive the automatic generation of models for testing.

Chapter 4

In this chapter we discuss techniques to test whether a system conforms to its model given in terms of an Input/Output Symbolic Transition System (IOSTS). IOSTSs are automata-based models using data types to enrich transitions with data-based messages and guards depending on state variables. We focus on symbolic execution techniques both to extract IOSTS behaviors to be tested in the role of test purposes and to ground test case generation.

Chapter 5

Using model-based approaches for designing embedded systems helps remove unnecessary details in a manner that reduces production costs, increases the potential for easy validation and verification, and facilitates reuse and evolution. In this context, a common practice is to use UML as the base language, possibly specialized by the so-called profiles. Despite the ever increasing number of profiles being built in many domains, there is still insufficient focus on discussing the issue of combining multiple profiles. Indeed, a single profile may not be adequate to cover all aspects required in the multidisciplinary domain of embedded systems.

In this chapter, we assess possible strategies for combining the SysML and MARTE profiles in a common modeling framework, while avoiding specification conflicts. We show that, despite some semantic and syntactical overlapping, the two are highly complementary for specifying embedded systems at different abstraction levels. We

conclude, however, that a convergence agenda is highly desirable to ensure proper alignment of some key language features.

Chapter 6

This chapter starts with a brief review of performance modeling formalisms and a discussion of the performance annotations that need to be added to UML software models in order to enable performance analysis. The principles for transforming annotated software models into performance models are then presented. Such model transformations must bridge a large semantic gap between the source and the target model; hence a pivot model is often used. An example of such a transformation is given, from UML extended with the MARTE profile to the Layered Queueing Network performance model. The role of an intermediate pivot language called Core Scenario Model is also discussed. The chapter ends with a discussion of the lessons learned and future challenges for integrating the analysis of multiple non-functional properties in the context of MDE.

Chapter 7

This chapter proposes to integrate design and formal modeling approaches, based on MARTE, IF and CDL, to evaluate different possible uses and configurations of a data acquisition software component. The uses are related to the actor's (sensor and application) behavior and configurations are related to implementation parameterization. Evaluation considers safety and performance properties and delay evaluation, which are automatically evaluated by the OBP tool. The work is illustrated using an example to show the impact of parameters and contextual use on software acquisition driver performances. Using this tool, it is possible to tune the driver's parameters to obtain the required

performances, in terms of delays, for a certain context use. The approach is applied to sensor monitoring applications.

Chapter 8

This chapter presents a new methodology for developing SoC/SoPC applications. This methodology is based on UML and MDD and capitalizes on the achievements of the “Electronic System Level” community by taking into account the new MARTE profile dedicated to real-time embedded systems. In the MOPCOM SoC/SoPC research project, a tooling has been developed to support this SoC/SoPC methodology, the MARTE profile, HDL code generation and documentation generation. A Cognitive Radio demonstrator is presented to illustrate the methodology and the tooling.

Chapter 1

Model Transformation: A Survey of the State of the Art

Rien ne se perd, rien ne se crée, tout se transforme.
(*Nothing is lost, nothing is created, everything is transformed*)
Antoine-Laurent de Lavoisier (1743-1794)

1.1. Model-driven engineering

Model-Driven Engineering (MDE) is an approach to software development where the principle artefacts are models (as opposed to source code). It is a natural next step in the evolution of software development to continue to raise the level of abstraction in order to tackle increasingly complex problems. The main goal is to reduce the *accidental complexity* [BRO 86] of software, caused by the technology, methods and programming languages used to develop software. Of course, the *essential complexity* that is inherent to the problem to be solved cannot be reduced, no matter which approach, technology or language is adopted.

Chapter written by Tom MENS.

The basic principle behind MDE is that *everything is a model*. As such, it provides a generic approach to deal with all possible software artefacts used and produced during the software development life-cycle (e.g. requirement specifications, analysis and design documents, test suites, source code, and so on). Even the languages used to specify the models can be considered as models too, which are referred to as *metamodels*.

The current *state-of-the-practice* of tool support for MDE is still in the *round-trip engineering* stage: the models and the code co-exist, and a change to either of the two artefacts requires a synchronization of the other. Ideally, this synchronization is automated, but in practice there is often some manual effort involved as well. In contrast, the *state of the art* in MDE support is *model centric*, where the code can be fully generated from the models [RAI 04].

Accepting the basic idea that everything is a model, and adopting a model-centric view, we need techniques and tools that allow us to manipulate and reason about such models. The technique that can be used to achieve this is commonly referred to as *model transformation*. According to [SEN 03, GER 02], model transformation is *the heart and soul of model-driven software development*. It is needed for supporting a wide range of model-driven activities such as code generation, model extraction, model refactoring, model verification, model simulation, and many more.

1.2. Model transformation

1.2.1. Definitions

Kleppe *et al.* [KLE 03] provide the following definition of model transformation: *a **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation***

***definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.*

This definition is very general, and covers a wide range of activities for which model transformation can be used: automatic code generation, model synthesis, model evolution, model simulation, model execution, model quality improvement (e.g. through model refactoring), model translation, model-based testing, model checking, model verification, and many more. For some types of activities we would like to support, the definition needs to be extended, in order to allow for model transformations that take more than one source model as input and/or produce multiple target models as output. The different source (resp. target) models do not even need to be described in the same modeling language. Examples of activities where we need more than one source or target model are model merging (in the context of collaborative modeling), model weaving and model composition [FLE 07, HID 09].

In order to support this variety of model transformation activities, we need to put in place a number of different mechanisms. Obviously, we need *transformation languages* that describe how to specify model transformations. This will be the topic of section 1.3. For those languages that have an underlying formal foundation, we need formal methods and theories to rely on. We also need *tools* that implement and support these languages and formalisms. A wide variety of such tools is available, research prototypes as well as commercial tools. Methodologies or processes are needed in order to help us to use all of these mechanisms in an efficient way. Examples are the Rational Unified Process (RUP, [KRU 03]) and the Executable UML methodology based on the

Schlaer-Mellor method [RAI 04]. To facilitate communication and interoperability, standards are needed for all of the above. The most obvious standards are those proposed by the OMG (e.g. UML, XMI, QVT, MOF, OCL, SysML and many more). Other *de facto* “standards” are those proposed by the Eclipse community (e.g. EMF, ECore, and so on).

1.2.2. *Taxonomy*

[MEN 06c] proposed a *taxonomy* of model transformation. Many of the ideas in this taxonomy were based on the discussions of a working group of a 2004 Dagstuhl seminar on Language Engineering for Model-Driven Software Development. We briefly review the essential parts of this taxonomy here.

Endogenous versus exogenous transformations

In order to transform models, these models need to be expressed in some modeling language (e.g. UML). A distinction can be made between *endogenous* and *exogenous* transformations, depending on the language(s) used to express source and target models involved in the model transformation. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages.

A typical example of an exogenous model transformation is *model synthesis*, in which a design model is extracted from source code. The inverse exogenous transformation is *code generation* to transform the design models into source code. Another well-known example of exogenous model transformation is *model translation*, in order to transform some representation of a model into an equivalent representation expressed in a different modeling language (e.g. UML to XMI, or class diagrams to entity-relationship diagrams).

A typical example of endogenous transformation is *optimization*: it aims to improve certain operational qualities (e.g. performance), while preserving the semantics of the model. A related endogenous transformation is *model refactoring*, which aims to improve the model structure.

Horizontal versus vertical transformations

An orthogonal way to classify model transformation is by looking at the *abstraction level* of its source and target models. For *horizontal* transformations, the source and target models must reside at the same level of abstraction. Typical examples are *model refactoring* (an endogenous transformation) and *model translation* (an exogenous transformation). For *vertical* transformations, the source and target models must reside at different abstraction levels. A typical example is *refinement*, where a specification is gradually refined into a full-fledged implementation, by means of successive refinement steps that add more concrete details [BAC 98].

1.3. Model transformation languages

Model transformation languages serve to specify the syntax and semantics of model transformations, and are essential if we want to provide automated support for model transformation. A wide variety of model transformation languages exist. Many of them have emerged from the academic community, while others originate from industry. In the latter category we find, for example, OMG's QVT specification [OBJ 08], which is compatible with the MDA approach based on MOF and UML. The *academic* languages include, without attempting to be complete: ATL, Kermeta, Tefkat, SiTra and many languages that are based on the underlying approach of graph transformation (e.g. ATOM3, AGG, Fujaba, GReAT, MOFLON, VIATRA2).

Due to this wealth of transformation languages, it is necessary to provide a taxonomy that allows us to assess the conceptual commonalities and differences between these languages. This is the purpose of the current section.

Declarative versus operational

A first criterion to compare transformation languages is whether they rely on a *declarative* or an *operational* (a.k.a. *imperative* or *constructive*) specification.

Declarative approaches focus on *what* needs to be transformed into what by defining a relationship or mapping between the source and target models. These approaches are attractive because they tend to be easier to write and understand by software engineers. In addition, desirable services such as source model traversal, traceability management and bidirectional transformations may be offered by the underlying transformation engine.

Operational approaches focus on *how* the transformation needs to be performed by specifying the steps that are required to derive the target models from the source models. Such approaches may be required to implement transformations for which declarative approaches fail to guarantee their services. Especially when the application order of a set of transformations needs to be controlled explicitly, an imperative approach is more appropriate thanks to its built-in notions of sequence, selection and iteration. Such explicit control may be required to implement transformations that reconcile source and target models after they have been both heavily manipulated outside the transformation tool.

Interestingly, the QVT specification [OBJ 08] offers two different languages: *QVT Relational* is a declarative transformation language, while *QVT Operational* belongs to the category of operational languages. Figure 1.1 shows an

example of the use of *QVT Relational*, while Figure 1.2 shows an example expressed in *QVT Operational*.

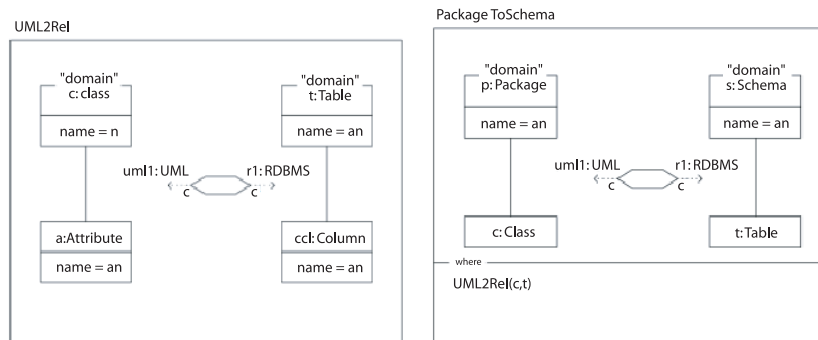


Figure 1.1. Part of the *Class2RDBMS* model transformation expressed using *QVT Relational*

```
transformation UML2RDBMS (in uml:UML, out rdbms:RDBMS) {
  // content of the transformation definition }
mapping Class:class2table() : Table when
{self.isPersistent()}
{ name := 't_' + self.name;
  column := self.attribute->map attr2column();
  key := self.map class2key(result.column);
}
mapping Attribute:attr2column() : Column
{ name := self.name;
  type := getSqlType(self.type);
}
mapping Class:class2key(in cols:Sequence(Column)) : Key
{ name := 'k_' + self.name;
  column := cols[kind='primary'];
}
```

Figure 1.2. Part of the *Class2RDBMS* model transformation expressed using *QVT Operational*

Textual versus visual

Another criterion to distinguish model transformation languages is how their concrete syntax is specified. *Textual* transformation languages (such as ATL and Kermeta)

require us to specify the model transformations using a textual description. *Visual* transformation languages (such as nearly all of the graph transformation languages) specify model transformations in a visual way.

Note that some transformation languages offer both alternatives. For example, for *QVT Relational* [OBJ 08], both a visual and a textual syntax is available. The visual syntax is illustrated in Figure 1.1, whereas the textual syntax is illustrated in Figure 1.2.

Other distinguishing characteristics

Many other criteria can be used to compare or distinguish model transformation languages. For example, we can distinguish between *general-purpose* and *domain-specific* transformation languages. We can also distinguish between languages that have been designed and implemented in an *ad hoc* way as opposed to languages that have a *formal* underlying foundation. As we will see later, the latter type of languages can be exploited to achieve some kind of formal analysis of the model transformations they represent.

Finally, the *expressiveness* of the transformation language is also very important. Ideally, the language should provide mechanisms to facilitate (de)composition and reuse of model transformations, the ability to specify higher-order transformations (transformations that can transform transformations), the ability to specify bidirectional transformations (a possibility that is offered by triple graph grammar approaches [GIE 06] such as MOFLON¹, and so on).

1.4. Model transformation activities

In this section, we will provide a brief overview, with references to relevant literature, of a wide variety of model-

1. <http://www.moflon.org>.

based activities in which model transformations are essential. While this overview is inevitably incomplete, it allows us to illustrate the importance and breadth of the field of model transformation.

Automatic code generation

Undoubtedly, *code generation* is one of the main motivations for using model transformation technology. It is used by various companies, including Airbus [SAB 09]. According to OMG's MDA approach [KLE 03], the goal is to transform platform-independent models (PIMs) into platform-specific models (PSMs) and ultimately to source code generated from these models. According to our transformation taxonomy in section 1.2.2, code generation is an example of a vertical, exogenous transformation.

Ideally, this type of transformation should be as automated as possible. Seen in this light, a promising approach is *Executable UML* [RAI 04]. It uses an action semantics language integrated into a well-defined subset of the UML to allow full code generation from models.

Model extraction

Model extraction is another example of a vertical, exogenous transformation. It is the inverse of code generation, and is an essential activity in reverse engineering and program comprehension. Taking the source code as input, it allows us to build a mental or visual model (e.g. a UML model) at a higher level of abstraction, in order to facilitate understanding of the code and how it is structured [MUR 01].

Model translation

Model translation is one of the horizontal, exogenous types of model transformation that has been used frequently in research literature for different purposes. In general, the goal is to transform a model, expressed in a particular

modeling language, into an “equivalent” model in another modeling language. A typical example of such a model transformation, that has been used frequently in research literature is the Class2RDBMS model transformation (see, e.g. [WIM 07, HID 09]). Its aim is to convert a class model (e.g. a UML class diagram) into a relational database model. This transformation is sometimes referred to as the object-relational mapping. It provides a bridge between the world of object-oriented specifications and relational database specifications. A partial example of this transformation, expressed using QVT, has been presented in Figures 1.1 and 1.2.

Another important application of model translation is to cope with the ill-defined, underspecified, semantics of some modeling languages (such as UML) by translating them into another semantic domain that has a sound formal semantics, so that we can apply some form of formal analysis to our models. [ANA 07] attempted to transform UML models into the formal language *Alloy*, and encountered several problems in doing so. [VAN 03] proposed a translation of UML models into the *description logics* formalism. [HER 08] translated UML class and sequence diagrams into the domain of graph transformation, thereby giving an operational and denotational semantics to a subset of UML.

A third reason for using model translation is to facilitate interoperability. We sometimes wish to use and interchange models between different modeling tools, and even between different modeling languages. A typical example is the translation of some tool-specific representations of UML models into XMI (and vice versa), OMG’s XML-based standard for model interchange [OBJ 07]. This facilitates exchanging UML models between different UML modeling tools.

Model simulation and execution

If we have a model that represents a certain behavior, we can use *model simulation* to actually “run” the model in order to validate if it behaves in the way we would expect it to. The strongest form of model simulation is probably *model execution*, in which we transform the model into a runnable system, either using interpreter or compiler technology. Executable UML is an approach that supports this [RAI 04].

Model simulation may sometimes require manual intervention, in order to provide essential information to the simulator that cannot be derived from the model itself. Model simulation may be very useful to execute multiple alternative execution paths, in order to explore various “what-if” scenarios, so that the most appropriate one can be chosen once we decide to implement the model.

In the context of UML, useful examples are the simulation of behavioral diagrams, such as activity diagrams and state machine diagrams. In some cases, to achieve such simulation, an intermediate model translation step may be required to transform the model into a domain with which we can associate an operational semantics. For example, we can simulate the behavior of a UML activity diagram by first translating it into a Petri net, and then executing this Petri net [STÖ 05]. An alternative would be to translate it into a graph transformation system and execute that system [ENG 08, ERM 05].

Model checking, verification and validation

As is the case for any software artefact, it is desirable to *verify* and *validate* models. *Model validation* corresponds to checking whether the model conforms to the requirements, i.e. it is a *useful* model. Approaches such as model simulation and model testing may be used to check this. *Model verification* analyses whether the model is correct, syntactically as well as semantically. *Syntactic analysis*

verifies whether the model is well-formed, i.e. whether it conforms to its metamodel and other constraints imposed by the modeling language. *Semantic analysis* can be done to verify dynamic or behavioral properties of the model, provided that the model is expressed in some formal semantic domain (possibly after having performed a model translation first).

A wide variety of model checking languages and tools exist (e.g. [CAB 07, VAR 04]), and the type of analysis we can do depends on the expressive power and properties supported by the associated formalism. For example, from a Petri net specification we can analyze properties such as reachability or deadlock freeness [PET 81]. From a graph grammar specification, we can analyze properties such as termination and confluence [HEC 02].

Next to model checking it is also necessary to validate and verify the model transformations themselves. Validation of model transformation allows us to assess whether the transformation is useful and meaningful. This can be achieved, for example, through *model transformation testing and verification* [NAR 08, BAU 09]. Verification of sets of model transformations is needed to ensure that they produce well-formed and correct models, and preserve (or improve) desirable properties such as (syntactical or semantical) correctness, consistency, and so on.

[STE 04] explain how to validate the declarative model transformation engine *Tefkat*. [VAR 04, KÜS 06] address the formal verification of model transformations expressed using (different variants of) graph transformation rules. Alternatively, [ANA 07] use the formal constraint solver and model checking language and *Alloy* tool to verify model transformations.

Model migration and co-evolution

Model transformations are also essential to cope with the inevitable evolution of models. In this context, an additional problem arises: not only do the models evolve, but so do the modeling languages in which these models are expressed [FAV 05]. With any change in the modeling language (e.g. a new version of UML that is introduced), model designers are confronted with the need to upgrade their models to this new version, or run the risk that their models will become obsolete or inconsistent. The activity of keeping models in sync with their evolving modeling languages is called *model co-evolution* or *co-adaptation*, and has been investigated by various authors [WAC 07, HER 09].

Note that the problem of model co-evolution is much larger than what is explained above. In fact, any software artefact that is directly or indirectly related to a model may need to co-evolve. A typical example of this is the round-trip engineering process, in which models and source code co-exist, and need to be synchronized whenever a change is made to the model or the source code [D'H 00].

Model quality improvement

A specific kind of model evolution for which transformations are particularly useful is *model quality improvement*. Models may have various types of quality criteria that need to be satisfied, but this quality tends to degrade over time due to the many changes that are made to the model during its lifetime. Therefore, we need transformations that allow us to improve the model quality. In particular, if we want to improve the structural quality of a model, we can make use of *model refactoring*. In analogy with *program refactorings* [FOW 99], it is an endogenous horizontal model transformation that improves the model's structure while preserving its behavior. Various authors have started to explore the problem of UML model refactoring [POR 03, ALE 04, MAR 05, Van 05, ZHA 05,

MEN 06a] so it is only a matter of time before UML modeling environments start supporting this activity.

Model inconsistency management

As a final example, the activity of *model inconsistency management* is also well-suited to being supported by model transformation. Due to the fact that models are typically expressed using multiple viewpoints [GRU 98], are under constant evolution, and are often developed in a collaborative setting, inconsistencies in models cannot be avoided. Therefore, we need techniques based on model transformation to repair such inconsistencies. [MEN 06b] propose to do this using graph transformation, while [VAN 03] propose an approach based on description logics. Other formalisms may be suited to support this activity as well.

1.5. Conclusion

To conclude this chapter, I hope to have convinced the reader of the omnipresence of model transformation in all areas and activities of model-driven engineering. Given its importance, it is not surprising that there are so many different types of model transformation languages and tools around. Every approach has its own specific merits and shortcomings, so it is quite important to choose the most appropriate approach for a given purpose. Hopefully, this chapter will help you to make an informed choice, or incite you to carry out research in this exciting area of model-driven engineering.

1.6. Acknowledgements

This Chapter was written in the context of the research project “Model-Driven Software Evolution”, an *Action de Recherche Concertée* financed by the *Ministère de la Communauté française – Direction générale de*

l'Enseignement non obligatoire et de la Recherche scientifique, Belgium.

1.7. Bibliography

- [ALE 04] ALEXANDRE CORREA C. W., “Applying Refactoring Techniques to UML/OCL Models”, *Proc. Int’l Conf. Unified Modeling Language*, vol. 3273 of Lecture Notes in Computer Science, Springer, October 2004, p. 173–187.
- [ANA 07] ANASTASAKIS K., BORDBAR B., GEORG G., RAY I., “UML2Alloy: A Challenging Model Transformation”, *Proc. Int’l Conf. Model Driven Languages and Systems*, vol. 4735 of Lecture Notes in Computer Science, Springer, 2007, p. 436–450.
- [BAC 98] BACK R.-J., VON WRIGHT J., *Refinement Calculus*, Springer, 1998.
- [BAU 09] BAUDRY B., GHOSH S., FRANCE R., LE TRAON Y., MOTTU J.-M., “Barriers to Systematic Model Transformation Testing”, *Communications of the ACM*, 2009, ACM.
- [BRO 86] BROOKS F. P., “No Silver Bullet—Essence and accidents of software engineering”, *Information Processing*, vol. 86, 1986, p. 1069–1076, Elsevier Science.
- [CAB 07] CABOT J., CLARISÓ R., RIERA D., “UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming”, *Proc. Int’l Conf. Automated Software Engineering*, 2007, p. 547–548.
- [D’H 00] D’HONDT T., DE VOLDER K., MENS K., WUYTS R., “Co-Evolution of Object-Oriented Design and Implementation”, *Proc. Int’l Symp. Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, January 2000.
- [ENG 08] ENGELS G., KLEPPE A., RENSINK A., SEMENYAK M., SOLTENBORN C., WEHRHEIM H., “From UML Activities to TAAL: Towards Behaviour-Preserving Model Transformations”, *Proc. European Conf. Model-Driven Architectures*, vol. 5095 of Lecture Notes in Computer Science, Springer, 2008, p. 94–109.

- [ERM 05] ERMEL C., HÖLSCHER K., KUSKE S., ZIEMANN P., “Animated simulation of integrated UML behavioral models based on graph transformation”, *Proc. Symp. Visual Languages and Human-Centric Computing*, IEEE Computer Society, 2005, p. 125–133.
- [FAV 05] FAVRE J.-M., “Languages evolve too! Changing the Software Time Scale”, *Proc. Int’l Workshop on Principles of Software Evolution*, IEEE Computer Society, 2005, p. 33-44.
- [FLE 07] FLEUREY F., BAUDRY B., FRANCE R. B., GHOSH S., “A Generic Approach for Automatic Model Composition”, GIESE H., Ed., *MoDELS Workshops*, vol. 5002 of Lecture Notes in Computer Science, Springer, 2007, p. 7-15.
- [FOW 99] FOWLER M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [GER 02] GERBER A., LAWLER M., RAYMOND K., STEEL J., WOOD A., “Transformation: The Missing Link of MDA”, *Proc. Int’l Conf. Graph Transformation*, vol. 2505 of Lecture Notes in Computer Science, Springer, 2002, p. 90–105.
- [GIE 06] GIESE H., WAGNER R., “Incremental Model Synchronization with Triple Graph Grammars”, *Proc. Int’l Conf. Model Driven Engineering Languages and Systems*, vol. 4199 of Lecture Notes in Computer Science, Springer, 2006, p. 543–557.
- [GRU 98] GRUNDY J. C., HOSKING J. G., MUGRIDGE W. B., “Inconsistency Management for Multiple-View Software Development Environments”, *IEEE Trans. Software Engineering*, vol. 24, num. 11, 1998, p. 960-981.
- [HEC 02] HECKEL R., MALTE KÜSTER J., TAENTZER G., “Confluence of Typed Attributed Graph Transformation Systems”, *Proc. Int’l Conf. Graph Transformation*, vol. 2505 of Lecture Notes in Computer Science, Springer, 2002, p. 161–176.
- [HER 08] HERMANN F., EHRIG H., TAENTZER G., “A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams”, *Electronic Notes in Theoretical Computer Science*, vol. 211, 2008, p. 261-269, Elsevier.

- [HER 09] HERRMANNSDOERFER M., BENZ S., JUERGENS E., “COPE: Automating Coupled Evolution of Metamodels and Models”, *Proc. European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, 2009.
- [HID 09] HIDAKA S., HU Z., KATO H., NAKANO K., “Towards a compositional approach to model transformation for software development”, *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, New York, NY, USA, 2009, ACM, p. 468–475.
- [KLE 03] KLEPPE A., WARMER J., BAST W., *MDA Explained, The Model-Driven Architecture: Practice and Promise*, Addison Wesley, 2003.
- [KRU 03] KRUCHTEN P., *The Rational Unified Process: An Introduction*, Addison-Wesley, 3rd edition, 2003.
- [KÜS 06] KÜSTER J. M., “Definition and validation of model transformations”, *Software and System Modeling*, vol. 5, num. 3, 2006, p. 233-259.
- [MAR 05] MARKOVIC S., BAAR T., “Refactoring OCL Annotated UML Class Diagrams”, *Proc. Int’l Conf. Model Driven Engineering Languages and Systems*, vol. 3713 of Lecture Notes in Computer Science, Springer, 2005, p. 280–294.
- [MEN 06a] MENS T., “On the Use of Graph Transformations for Model Refactoring”, *Generative and Transformational Techniques in Software Engineering*, vol. 4143 of Lecture Notes in Computer Science, Springer, 2006, p. 219-257.
- [MEN 06b] MENS T., VAN DER STRAETEN R., D’HONDT M., “Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis”, *Proc. Int’l Conf. Model Driven Engineering Languages and Systems*, vol. 4199 of Lecture Notes in Computer Science, Springer, October 2006, p. 200-214.
- [MEN 06c] MENS T., VAN GORP P., “A Taxonomy of Model Transformation”, *Proc. Int’l Workshop on Graph and Model Transformation (GraMoT 2005)*, Electronic Notes in Theoretical Computer Science, Elsevier, 2006.

- [MUR 01] MURPHY G. C., NOTKIN D., SULLIVAN K. J., “Software Reflexion Models: Bridging the Gap between Design and Implementation”, *IEEE Transactions on Software Engineering*, vol. 27, num. 4, 2001, p. 364-380, IEEE Computer Society.
- [NAR 08] NARAYANAN A., KARSAI G., “Towards Verifying Model Transformations”, *Notes in Theoretical Computer Science*, num. 211, 2008, p. 191–200, Elsevier.
- [OBJ 07] OBJECT MANAGEMENT GROUP, “XML Metadata Interchange (XMI) version 2.1.1”, formal/2007-12-01, December 2007.
- [OBJ 08] OBJECT MANAGEMENT GROUP, “Query/View/Transformation Specification version 1.0”, formal/2008-04-03, April 2008.
- [PET 81] PETERSON J. L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.
- [POR 03] PORRES I., “Model Refactorings as Rule-Based Update Transformations”, STEVENS P., WHITTLE J., BOOCH G., Eds., *UML 2003 - The Unified Modeling Language*, vol. 2863 of Lecture Notes in Computer Science, Springer, 2003, p. 159-174.
- [RAI 04] RAISTRICK C., FRANCIS P., WRIGHT J., CARTER C., WILKIE I., *Model Driven Architecture with Executable UML*, Cambridge, 2004.
- [SAB 09] SABATIER L., POUPART E., DALBIN J.-C., BAZEX P., LE THI T.-T., MILLIAN T., “Transformation de modèles pour des applications aéronautiques et spatiales: vérification de propriétés”, *Journées NEPTUNE’2009*, 2009.
- [SEN 03] SENDALL S., KOZACZYNSKI W., “Model Transformation: The Heart and Soul of Model-Driven Software Development”, *IEEE Software*, vol. 20, num. 5, 2003, p. 42–45, Special Issue on Model-Driven Software Development.
- [STE 04] STEEL J., LAWLEY M., “Model-Based Test Driven Development of the Tefkat Model-Transformation Engine”, *International Symposium on Software Reliability Engineering*, vol. 0, 2004, p. 151-160, IEEE Computer Society.

- [STÖ 05] STÖRRLE H., HAUSMANN J. H., “Towards a Formal Semantics of UML 2.0 Activities”, LIGGESMEYER P., POHL K., GOEDICKE M., Eds., *Software Engineering*, vol. 64 of LNI, GI, 2005, p. 117-128.
- [VAN 03] VAN DER STRAETEN R., MENS T., SIMMONDS J., JONCKERS V., “Using Description Logics to Maintain Consistency Between UML Models”, *Proc. Unified Modeling Language*, vol. 2863 of Lecture Notes in Computer Science, Springer, 2003, p. 326–340.
- [VAN 05] VAN KEMPEN M., CHAUDRON M., KOUDRIE D., BOAKE A., “Towards Proving Preservation of Behaviour of Refactoring of UML Models”, *Proc. SAICSIT 2005*, 2005, p. 111-118.
- [VAR 04] VARRÒ D., “Automated formal verification of visual modeling languages by model checking”, *Software and Systems Modeling*, vol. 3, num. 2, 2004, p. 85-113, Elsevier.
- [WAC 07] WACHSMUTH G., “Metamodel Adaptation and Model Co-adaptation”, *Proc. European Conf. Object-Oriented Programming*, vol. 4609 of Lecture Notes in Computer Science, Springer, 2007, p. 600–624.
- [WIM 07] WIMMER M., STROMMER M., KARGL H., KRAMLER G., “Towards Model Transformation Generation by Example”, *Proc. 40th Hawaii Int’l Conf. System Sciences*, IEEE Computer Society, 2007.
- [ZHA 05] ZHANG J., LIN Y., GRAY J., “Generic and Domain-Specific Model Refactoring using a Model Transformation Engine”, *Model-driven Software Development – Research and Practice in Software Engineering*, Springer, 2005.

Chapter 2

Model-Based Code Generation

2.1. Introduction

The benefits of an implementation-free model, fashionably named Platform Independent Model (PIM), are now widely recognized. However, the space and speed constraints characteristic of many embedded systems means there is necessarily a significant distortion of the PIM when deriving a Platform-Specific Implementation (PSI). This leads to the situation where maintenance of the PIM in the face of changes to the PSI becomes very costly, and in many cases is abandoned, leaving the PIM to fall into obsolescence. It is then the PSI that must be maintained in the face of requirement changes. The problem with the PSI is that it is many times larger and more complex than the PIM, and is correspondingly more expensive to maintain. Also, it is obviously platform-specific, so migration to a new platform, where different optimizations may be required, becomes increasingly difficult.

Chapter written by Chris RAISTRICK.

What we need is a process that allows us to maintain a clear separation between the specification of required behavior, and the specification of how that behavior should be realized in an optimized way on a specific platform. This paper describes such a process.

2.2. The model-driven architecture (MDA) process

MDA defines two primary types of model: the Platform Independent Model (PIM) and the Platform Specific Model (PSM). Here the term *platform* is used to refer to technology and engineering details that are irrelevant to the fundamental functionality of the software.

These model types are a key concept in MDA; it mandates the separation of concerns of analysis (the PIM) from its realization on a particular computing platform and technology (the PSM) and recognizes that the refinement relationship between the two types of model should be achieved by applying a *mapping*. It goes on to state that such a mapping may be used to realize many such refinement relationships between different PIMs and their corresponding PSMs (i.e. they can be reused); furthermore, it states that the mapping can be captured as a model itself, expressed in UML. It also recognizes that the PIM to PSM mapping may be fully automated if we define both the PIM and the mapping with sufficient rigor.

The MDA process can be seen as a framework with a number of embedded principles. It can be applied in such a way that code is generated manually or automatically. As manual approaches to code generation have been exhaustively documented for over three decades, this paper will concentrate on sophisticated automatic code generation.

2.3. The automated approach to code generation

MDA encourages the designers to formalize the rules about how to transform models into code. This means that all components are generated in a consistent and comprehensible way. We shall say more about how the translation rules are formalized later.

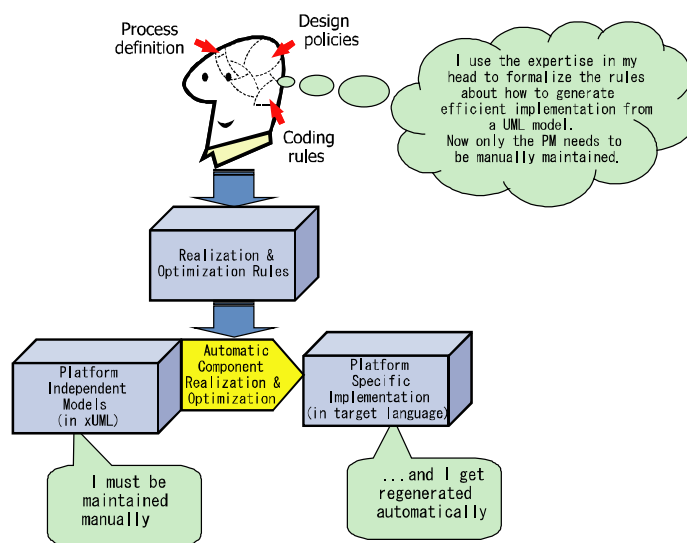


Figure 2.1.

The automated approach has the advantage that only the PIM needs to be maintained, avoiding the tendency for specification and implementation to become out of step.

Let us consider the automated approach in more detail.

Figure 2.2 outlines the process of MDA with xUML, in which:

- The domain experts build precise, PIMs of the various aspects of the system. These models embody no assumptions about the target execution platform, which makes them

smaller and simpler, and consequently less expensive to construct and test. Use of Executable UML (xUML) to express the models means that they can be verified in a simulation environment prior to implementation;

– the verified PIMs are automatically transformed into a Platform-Specific Implementation in the desired target language.

With this strategy, all the business intellectual property is captured, in reusable form, as xUML models. Because application knowledge is held in PIMs, its longevity is guaranteed, as it will not become obsolete when today's technologies are superseded.

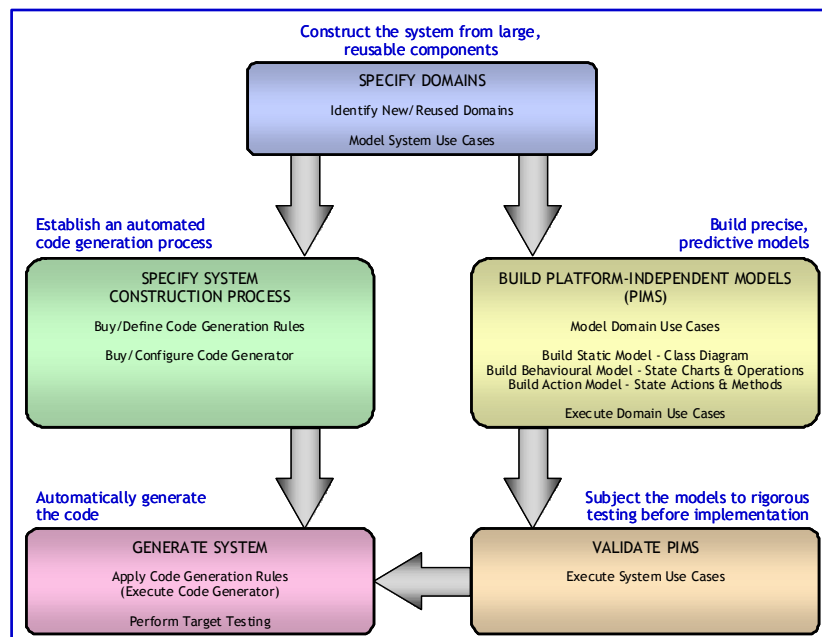


Figure 2.2. The MDA process and work products

By embodying these principles, use of MDA with xUML raises software engineering to a level comparable with more

mature engineering disciplines, such as electronic engineering, aeronautical engineering and civil engineering. These engineering disciplines are characterized by the way that engineers *routinely*:

- construct the product from *large reusable components*;
- build precise, predictive models;
- subject the models to rigorous testing prior to implementation;
- establish a well-defined, highly automated construction process.

The following sections describe the process and its artifacts in more detail.

2.4. Domain modeling

Domain-based or subject matter-based partitioning is today a widely accepted and mature partitioning strategy. The basis for domain partitioning is to recognize that any system comprises a set of subject matters, or *domains*. These units will be used as a basis for partitioning the analysis effort for the system. A domain will comprise a number of classes and can be represented in UML using a *package*.

Figure 2.3 below shows a *domain chart* for a simplified air traffic control system. Each domain (shown as a UML package) on the chart represents a distinct subject matter within the whole system, while the dotted arrows represent *dependencies* between the domains. A dependency indicates that the client requires some services that are fulfilled by the server domain. Each domain will be specified in the form of a PIM.

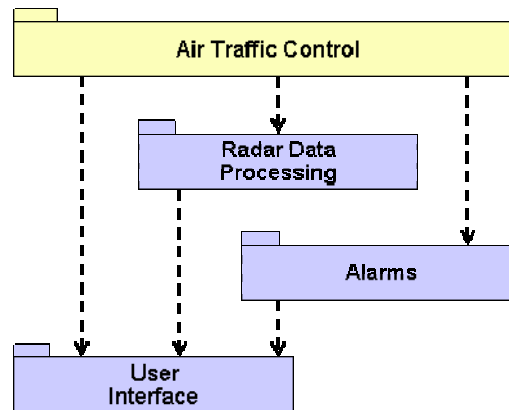


Figure 2.3. A simplified domain chart for an air traffic control system

A domain is defined as “a separate real, hypothetical or abstract world inhabited by a distinct set of classes that behave according to the rules and policies characteristic of that domain”.

Sounds intimidating? We can unpack the definition into smaller components:

“A *separate real, hypothetical or abstract world*” means that the domain might represent a “*real world*”, such as Air Traffic Control, Military Command and Control or Patient Administration for instance. Such a domain usually reflects the end user requirements directly. We typically have little or no discretion over these domains; we just formalize the requirements that impinge upon them, aided by any use cases or other requirements that have been documented.

A “*hypothetical world*” might be a domain that performs mathematical transformations, such as 3D Geometry or Statistical Analysis. Such domains serve to formalize the rules of mathematics – again, not much scope for imagination here. An “*abstract world*” is a world that we have invented for our own convenience, such as a User

Interface or Alarms domain. In these domains, the requirements are invented by us to meet the overall needs of the system. For example, we need to establish a policy regarding unavailable menu items; are they grayed out or are they not shown?

“A *distinct set of classes*” means that a class should appear in only one domain. Note, though, that the same real world thing can appear at different levels of abstraction on different domains. For example, a real aircraft might appear as an Aircraft, Freight Carrying Vehicle, Serviceable Item, Radar Track and Icon in various domains. These are known as *counterpart classes*.

“Behave according to the rules and policies characteristic of the domain” means that each class understands the context within which it exists. An Aircraft class in the Air Traffic Control domain embodies air traffic control rules about separation and so forth. It knows nothing about how it is displayed and its behavior is only modeled from the viewpoint of its containing domain.

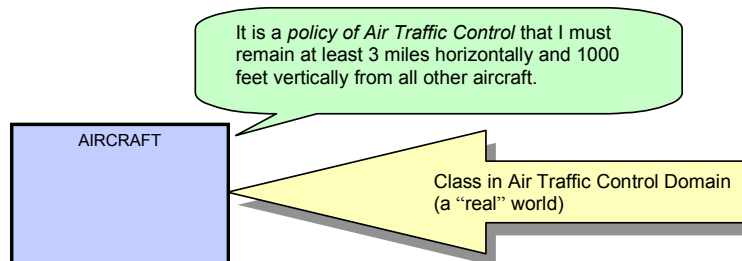


Figure 2.4. A class from the air traffic control domain

An *Icon* in a *User Interface* domain knows only about policies for displaying icons. It knows nothing about the rules and policies governing the things it represents. It is this clean separation of concerns that is the hallmark of

domain partitioning; it leads to simpler classes that are easier to test and reuse.

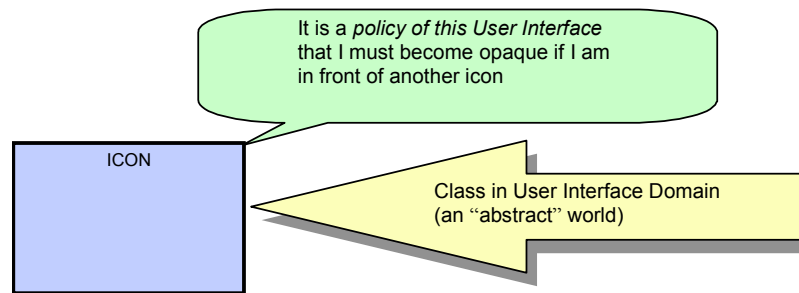


Figure 2.5. A class from the user interface domain

This approach exhibits a number of distinct advantages:

- *Reuse*: each domain is largely self contained, forming a potentially large reusable component.
- *Well defined interfaces*: a domain presents a well-defined, contractual interface to other domains which may want to use its services.
- *Effective utilization of subject matter knowledge*: each domain is a subject matter in its own right, therefore subject matter experts may analyze the appropriate domain (or set of domains), unhindered by consideration of other knowledge areas unfamiliar to them.
- *Stability to changing requirements*: the domain that captures the purpose of the system, with regard to the end users point of view, is the application domain; it will typically be augmented as further requirements are modeled, whilst domains further down the domain chart are isolated from such changes.
- *Stability to changing technology*: as technology advances inevitably parts of a system will become obsolete. To avoid this we must ensure that new technologies are readily

incorporated into the framework of the system. Domain partitioning recognizes this issue and allows service domains (those that represent highly reusable, technology-oriented subject matters) to be replaced in a highly modular fashion, without affecting other domains.

– *Incorporation of third party software*: many systems will incorporate legacy code, third party libraries or services. Domain partitioning recognizes that this is a risk area for any project and defines such units as implementation domains. The domain approach therefore does not insist upon a homogeneous approach, but rather manages the interfaces between differing components, promoting the use of COTS (Commercial Off-The-Shelf) software where appropriate.

– *Incorporation into a use case driven approach*: interaction diagrams are used to document use cases in terms of the domain level interactions. A high-level view of system interactions is extremely valuable and can be used to help scope each domain and define its interfaces.

These are the principles that underpin the MDA process; we separate our system into subject matters, some application-oriented, some representing pervasive services such as communications and persistence, and some technology-oriented.

A PIM is built for each domain, with well-defined interfaces. Each PIM is uncontaminated by other subject matters makes it both simple and reusable.

2.5. The executable UML (xUML) formalism

The Unified Modeling Language (UML), as its name suggests provides a unified notation for the representation of

various aspects of object-oriented systems. Let us examine the contrast between UML and xUML.

UML specifies a diagrammatic language that allows systems to be specified using a number of diagram types; however it is quite informal about how the different diagrams are to be used. For example, state machines can be used to describe use cases, subsystems and objects. This means that a reader must first establish the context for the diagram they are reading before being able to understand it.

In xUML, notations are used for a specific purpose: a state chart is always associated with a class, so a state machine always describes the behavior of an object.

In summary, xUML is a simple, precise subset of UML, specifically designed to allow construction of rigorous PIMs. The key facets of xUML are illustrated in Figure 2.6, which shows that:

- each system is partitioned into *domains*, representing areas of expertise;
- each domain is partitioned into classes, which together will fulfill the data and processing requirements of each domain;
- each class can have a *state machine*, which processes asynchronous signals directed to that class by executing state actions;
- each class can have *operations*, which perform synchronous processing.

The state actions and operation methods are specified using a UML Action Language to preserve platform independence.

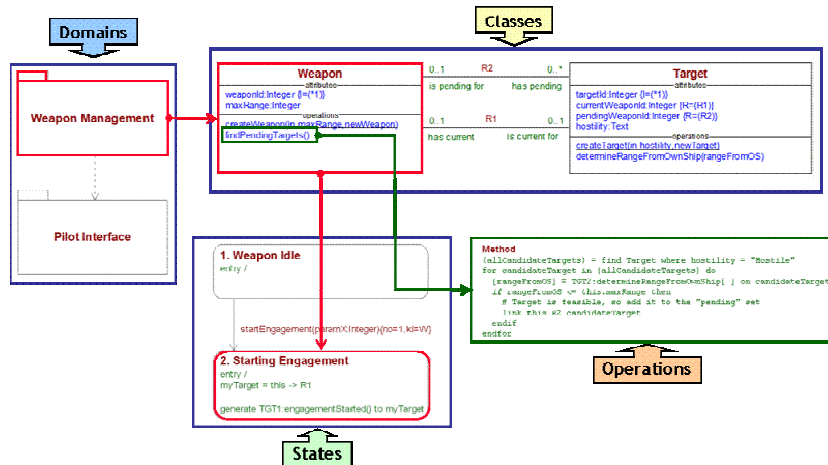


Figure 2.6. xUML model layers

There are many benefits that accrue from use of a precise UML subset, including reduced learning costs and less idiosyncratic models, but those that impinge directly on system generation are:

- there are fewer translation rules to define, reducing the cost of building a code generator;
- the runtime overheads, especially those associated with state machines can be drastically reduced.

2.6. System generation

This section describes a strategy by which users can create code generators to generate code that meets their exact specification. The overall process of xUML modeling and code generation is summarized in Figure 2.7 below.

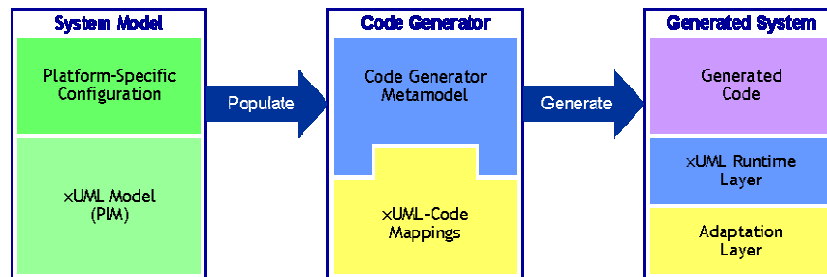


Figure 2.7. *Modeling and code generation*

A typical code generator suite consists of:

- **Code Generator**: a set of metamodels that form the basis for translating xUML models into code;
- **xUML Runtime Layer**: a portable run-time library. This provides the xUML “Virtual Machine” upon which the generated code will execute. In some embedded target architectures, the runtime is either very thin or non-existent, allowing generation of very compact implementations;
- **Adaptation Layer**: a non-portable, platform-dependent run-time library. This maps the run-time layer onto the underlying operating system and middleware. The interface to the Adaptation Layer is well-defined, allowing users to build their own, if an off-the-shelf implementation for any specific operating system is unavailable;
- **xUML-Code Mappings**: a set of language-specific mappings, typically bought off-the-shelf, specifying the rules for translating xUML models into that language. These can be configured by users if required to achieve specific target code qualities. Alternatively, users can define their own complete set of xUML-Code mappings to meet their particular needs.

The metamodel framework incorporates all the standard processing needed for any code generator, such as checking

the models' structure and parsing the action language. The yellow shaded components in Figure 2.7 are those typically configured by users to address specific requirements for their target system.

The code generator itself is a set of domain models expressed using xUML. The domains represent the various components of an xUML system, and the classes represent the elements that make up those components. Figure 2.8 below shows that the “Executable UML” domain (or formalism) contains the notions “Domain”, “Class” and “Attribute”. Each element contains operations which specify how to map that xUML element onto a specific target language. Different action language methods can be embedded within these models to generate code in different target languages with different static and runtime characteristics. For example, the class “Class” has an operation named “generateCode”, for which many rival methods can be specified to map a UML class to C, C++, Java, Ada or any other language as required.

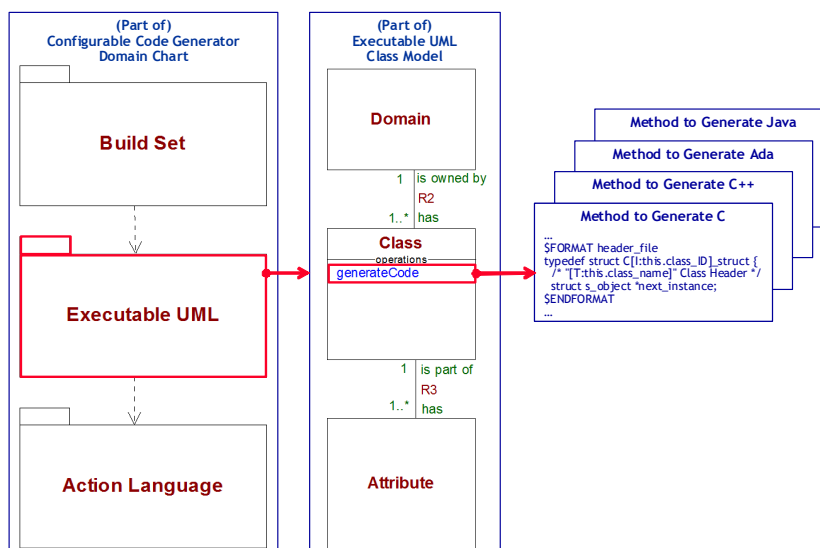


Figure 2.8. Code generator model structure

2.7. Executable UML to code mappings

Although the general MDA process permits multiple transformations from PIM to PSM to PSI, we will look initially at the idea of transforming PIMs expressed in executable UML directly into a PSI (i.e. code). That is, we will look at the construction of an executable UML code generator. Given an xUML model, there are a large number of ways that we might generate code. Some issues are illustrated in Figure 2.9 below, which shows some candidate design decisions regarding target language, software organization paradigm, memory management strategy, scheduling strategy, distribution scheme, persistence mechanisms and so on.

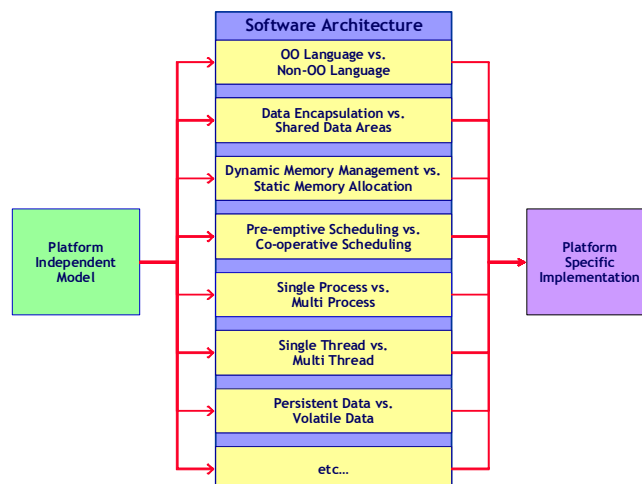


Figure 2.9. *Software architecture decisions*

For the purposes of code generation, we will refer to the particular choice of all these factors as the “Software Architecture” or simply “Architecture” for the system.

The code generator to be used for a particular architecture must provide the ability to translate from xUML models to

target code. Figure 2.10 below shows an example of the translation of a fragment of an xUML model into target code in a hypothetical architecture. For the remainder of this Chapter, we will refer to this example architecture as HA-1. HA-1 is a single task, ‘C’-based architecture. Note that this approach does not assume use of an object-oriented programming language, which in an embedded system can impose unwelcome overheads. In HA-1, the translation rules that have been developed require that objects of each class are held as a linked list of structures.

The structure members are derived both from the xUML model and from the fixed requirements of the architecture itself. In the lower box of the figure, we can see the translated structure definition for the “Target” class in the xUML model fragment. This structure has the following features:

- a unique name for the structure type derived from the number of the domain model (43) that the “Target” class resides in, and from the number of the “Target” class itself (3);
- various generic structure members (`next_instance`, `prev_instance`, `rel_ptr` and `cpr_ptr`) which are memory pointers for maintaining the linked list of objects as well as pointers to the intra-domain and inter-domain (counterpart) relationships that the object is currently participating in;
- one structure member for each attribute of the “Target” class in the xUML model, the type of each of which is derived from the type stored in the xUML model;
- since this is an active class (i.e. it has a state machine), a structure member holding the current state of this object “Target”. The type of this member is an enumeration, the members of which must be derived from the set of states defined in the “Target” state model;

– to aid readability of the generated code, a comment has been placed in the structure definition indicating that it has been generated from the “Target” class.

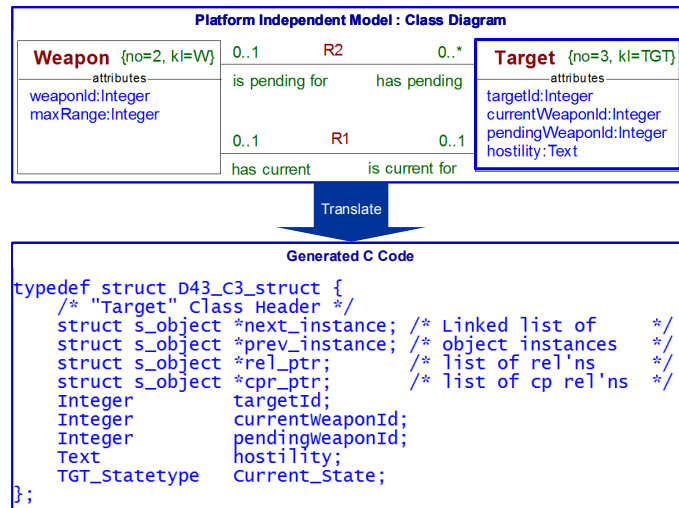


Figure 2.10. Example PIM to code mapping

In a later version of HA-1 perhaps, we would wish to improve both the performance and size of the generated code. Some of the options that we might consider are:

- remove the “currentWeaponId” member of the D43_C3 (“Target”) structure, because this referential attribute is never read at run time;
- for classes with a “Static Population” tag attached to them, use a statically allocated array of structures to hold the instance data. This would require that for such tagged classes the “next_instance” and “prev_instance” members would not appear in the generated structure definition;
- for classes (such as “Target” in this example) that have only single valued associations attached to them, replace the pointer to a linked list of relationship pointers with a single

pointer to (in this case) the current and pending “Weapon” instances;

– combine pairs of classes with mandatory one-to-one relationships between them with a single structure definition incorporating the attributes of both classes. This would involve:

- changed generation of the structure definition,
- changed code generation from all the ASL that accessed instances and attributes of the classes concerned,
- changed code generation for any association manipulation,
- reconciliation with any class/instance dispersal strategies if HA-1 was to become distributed,
- management of the case where both combined classes have state models.

From this discussion we can see that a code generator may be called upon to perform very subtle and complex mappings from xUML models into code. The execution of such mappings might involve taking information from a wide variety of sources within the model and making complex decisions about the best route for generation. To clarify what is going on inside the translator as it executes; the steps involved in generating code from a PIM are described below.

Step 1: Build the PIM

The domain experts build PIMs using xUML, such as this one:

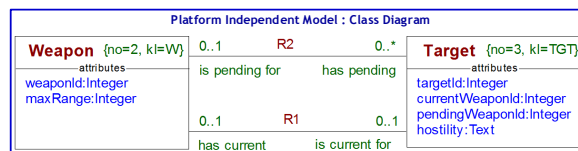


Figure 2.11. Example platform independent model class diagram

Step 2: Populate the metamodels

We first consider the “Executable UML” domain. As shown in Figure 2.12 below, this domain contains classes such as “Domain”, “Class” and “Attribute”. This figure also shows examples of the instances of these classes when the domain is populated with the example PIM (“Weapon – Target”) that was introduced in Figure 2.11 above.

We can see that our PIM has one domain, two classes and a total of eight attributes (including the two “Current_State” attributes as both classes have a state machine), and these manifest themselves as objects in the populated metamodel. We use a Populator that instantiates the metamodels with this information obtained from the PIM.

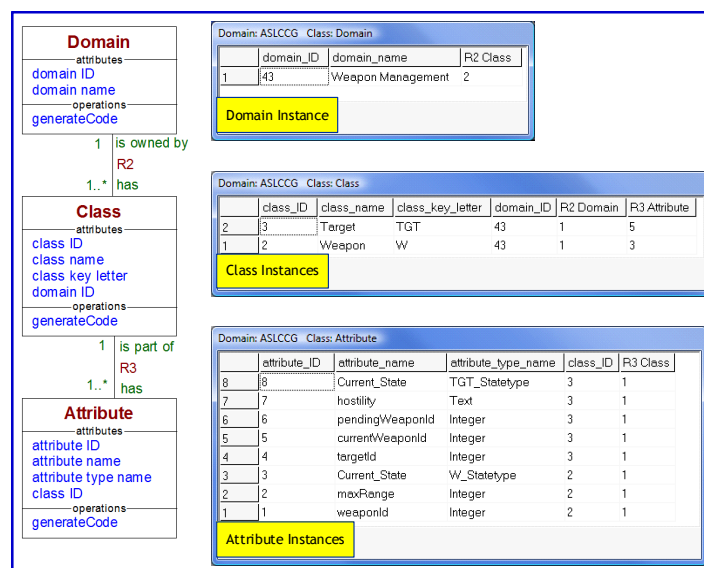


Figure 2.12. Example populated executable UML metamodel

Step 3: Generate the code

The task of translation involves iterating through these instances and generating suitable code from them.

Figure 2.13 below shows part of a translation engine, written in ASL, which generates the first part of the class headers for our hypothetical architecture HA-1.

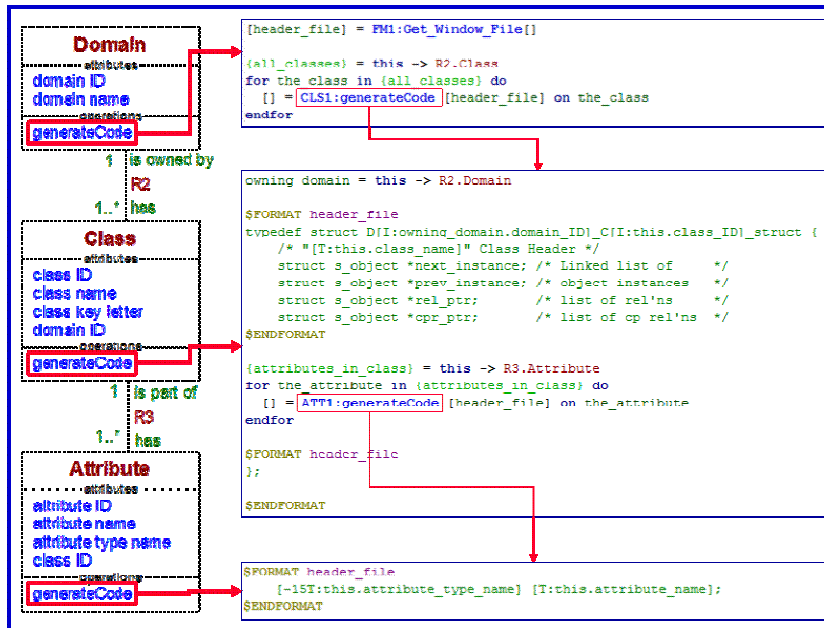


Figure 2.13. Example class model code generation rules

The operation of this code generator is summarized below:

- Generation of the code for the domain is initiated by calling operation “generateCode” on the solitary “Domain” object.
- The Action Specification Language (ASL) in the “Domain.generateCode” method (in the top box in Figure 2.13) finds the set of all the classes in this domain by navigating the association R2 in the “Executable UML” domain. The ASL then iterates over all the instances of “Class” in the set and for each instance invokes the operation “generateCode” on the “Class” class.

– The ASL in this “Class.generateCode” method (in the middle box in Figure 2.13) generates the first part of the structure definition for HA-1. This operation accesses attributes of the “Domain” and “Class” classes to generate the header code. The actual code construction is carried out by the \$FORMAT block which is an ASL feature that allows modelers to perform string manipulation. The construct takes the literal text between the \$FORMAT and \$ENDFORMAT markers and outputs it to the entity called “header_file” which is a data item of type “File”. Within the literal text of the \$FORMAT block there are embedded substitutions delineated by use of square brackets “[....]”. These specify that the value of the variable whose name is shown in the brackets should be substituted into the text. The characters before the colon (“:”) are format specifications.

– Once the class-specific part of the header file has been generated, the method finds the set of all attributes for this class by navigating association R3 in the “Executable UML” domain. The ASL then iterates over all instances of “Attribute” in the set, and for each instance invokes the operation “generateCode” on the “Attribute” class.

– The ASL in the “Attribute.generateCode” method (in the bottom box in Figure 2.13) accesses the attributes “attribute_name” and “attribute_type_name” of the “Attribute” instance to format a structure member declaration. In HA-1 the mapping between ASL types and “c” types is achieved via a header file that contains fixed typedefs for all the ASL types.

We have illustrated here the part of the translator that generates the data structures from the class model. The process of generating code from the action language is exactly the same, and based upon populating and translating instances in the “Action Language” domain, a fragment of which is shown in Figure 2.14 below.

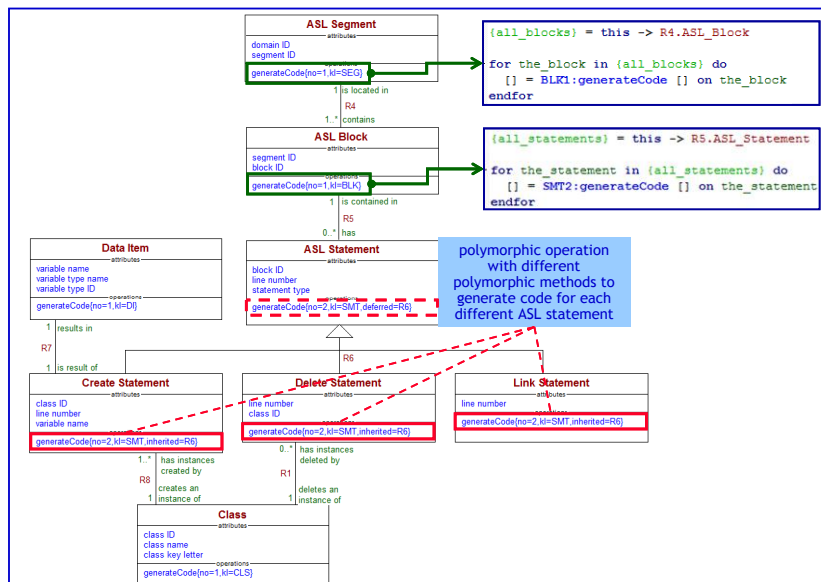


Figure 2.14. Example action language code generation rules

In this model, code generation for each “ASL Segment” (typically a state action or operation method) is initiated by invoking “generateCode” on the “ASL Segment” object. This in turn finds all linked “ASL Block” objects across R4 and invokes “generateCode” on each of these. These in turn find all linked “ASL Statement” objects across R5, and invoke “generateCode” on each of these. Note that the “generateCode” operation of the “ASL Statement” class is polymorphic, allowing us to implement rival versions of this method for each type of ASL statement, represented as the subclasses “Create Statement”, “Delete Statement” and so on.

2.8. Conclusions

We have shown how a configurable code generation framework can be used to exploit the full power of the MDA

approach. xUML is used to capture abstract platform models, and system generation is achieved by creating ASL that iterates over the information captured by these models.

Using this approach, developers can separate platform independent aspects of the system from specific platform details thus realizing the benefits of:

- reusable platform independent models;
- simpler, easier to understand, views of the system;

while avoiding the pitfalls of:

- the laborious creation of elaborated platform specific models;
- the triple redundancy involved in maintaining separate PIMs, PSMs and PSIs.

The strategy enables very sophisticated code generators to be realized, enabling developers to address the needs of the most demanding real-time embedded systems.

2.9. Bibliography

[RAI 04] RAISTRICK, C. *et al.*, *Model Driven Architecture with Executable UML*, Cambridge University Press, 2004.

Chapter 3

Testing Model Transformations: A Case for Test Generation from Input Domain Models

3.1. Introduction

Model transformation is a key mechanism when building distributed real-time systems (DRES) with model-driven development (MDD). It is used to automatically perform a large number of tasks in the development of DRES. The DOC group at Vanderbilt University has extensively investigated MDD for DRES. In this context, Madl *et al.* [MAD 06] use model transformations in order to apply model checking techniques on early design models, Gokhale *et al.* [GOK 08] develop model transformations that automate deployment tasks of component-based DRES and Shankaran *et al.* [SHA 09] use model transformations to dynamically adapt a DRES when its environment changes. The ACCORD/UML [GER 00] methodology developed by CEA also makes an extensive use of model transformations for a

Chapter written by Benoit BAUDRY.

model-driven development of DRES. Model transformations encapsulate specific steps in the development methodology and generate optimized code. Airbus develops large model transformations that automatically generate optimized embedded code for the A380 from SCADE models.

Due to the critical role that model transformations play in the development of DRES, thorough validation techniques are required to ensure their correctness. A fault in a transformation can introduce a fault in the transformed model, which, if undetected and not removed, can propagate to other models in successive development steps. As a fault propagates further, it becomes more difficult to detect and isolate. Since model transformations are meant to be reused, faults present in them may result in many faulty models. Several studies have investigated static verification techniques for model transformations. For example, Küster [KUS 06] focuses on the formal proof of the termination and confluence of graph transformation, and Anastasakis *et al.* [ANA 07] analyze properties on a formal specification of the transformation in alloy.

In this paper we are interested in adapting software testing techniques to validate model transformations. In particular, we focus on the generation and qualification of test data for model transformations. To test a model transformation, a tester will usually provide a set of *test models*, run the transformation with these models and check the correctness of the result.

While it is fairly easy to provide some input models, qualifying the relevance of these models for testing is an important challenge in the context of model transformations. As for any testing task, it is important to have precise adequacy criteria that can qualify a set of test data [BAU 09].

Model transformations specify how elements from the source metamodel are transformed into elements of the target metamodel. The source metamodel completely specifies the input domain of the transformation: the set of licit input models. In this context, the idea is to evaluate the adequacy of test models with respect to their coverage of the source metamodel. For instance, test models should instantiate each class and each relation of the source metamodel at least once. In the following we present test adequacy criteria based on the coverage of the source metamodel. We also discuss the automatic generation of test models that satisfy these criteria.

Before presenting the specific generation of test data for model transformation, we recall general techniques and current challenges for test generation from a model of the input domain. We briefly introduce category-partition testing [OST 88] and combinatorial interaction testing [COH 97] as two black-box techniques for the systematic selection of a subset of values in large domains. These techniques are a specific case of *model-based testing*.

Utting *et al.* [UTT 07b] identify four different approaches to model-based testing: generation of test data from a domain model, generation of test cases from an environmental model, generation of test cases with oracle from a behavior model, generation of test scripts from abstract tests. Utting *et al.*'s book focuses mainly on the third approach, while in this Chapter we will introduce techniques related to the first approach. Ammann *et al.* [AMM 08] propose another classification of structures from which it is possible to design test cases: graphs, logic, input domain, syntax. According to this taxonomy, the techniques introduced in this lecture are related to the last two structures: design of test data from an input domain model and from a model of the syntax (e.g. the source metamodel for a model transformation).

3.2. Challenges for testing systems with large input domains

One important aspect of the growing complexity of software systems is that these systems tend to be increasingly open to their environment. In particular, this means that many systems can operate on a very large amount of information provided by the user and/or offer mechanisms for dynamic reconfiguration. In both cases, these systems are characterized by a very large domain on which they have to run. It is usually not possible to test these systems with all possible input and in all possible configurations. The challenge for test data generation is to propose criteria to systematically select a subset of data that will still ensure a certain level of trust in the system being tested.

In this section we present several examples where such issues occur for test generation.

3.2.1. *Large set of input data*

The first category of systems that has a large input domain is the set of all programs that process a large set of data. These data can be provided by other software components or by users. Examples of these systems are all the web applications that process user input provided through a form.

Figure 3.1 displays an example of such a form that a user must fill in order to register with a conference online. On this simple, very common form, there are 18 variables. Some of these variables can take an infinitely large number of values (all the fields that require a String value such as address, name, etc.), and some others have a finite domain: the combobox for states defines 72 values, 228 values for country, 4 values for special needs and a binary value for

IEEE contact. In addition to the large domains for each variable, the global input domain for this page is the total number of combinations of values for each variable. This number is $72 * 229 * 4 * 2 * 14^{\text{#String values}}$. It is important to note that there exist some constraints between the fields that reduce the number of combinations. For example, if the country is neither Canada nor the USA, there is no need to provide a value for the Province/State field. In order to test this registration system, it is necessary to select a subset of all possible input values. In particular, it is necessary first to reduce the set of all possible String values to a finite set of test data; and second to select a small number of configurations of data.

Instructions

- ◆ red asterisk (*) indicated required field
- ◆ Type the first letter of each name in capital letters. (ex. John Doe - NOTE: john doe/JOHN DOE is not

General Information

* First/Given Name	
* Last/Family/Surname	
Organization	
* Address Line 1	
Address Line 2	
* City	
US State/Canadian Province	-- Select One -- v
* Country/Region	--Select a Country-- v
Zip/Postal Code	
* Primary Contact #	
Fax #	
* Email	
Additional Email	
A registration confirmation will be forwarded to the address(es) entered.	
Emergency Contact	
Emergency Contact #	
Dietary Restrictions	
Please specify the foods you cannot eat.	
Special Needs	No v

IEEE may use the information you provide to contact you from time to time concerning conferences, technical products and services or to ask your opinions.

May IEEE contact you? Yes

Figure 3.1. An example of a large domain: a web form

3.2.2. Configurable systems

Highly configurable and adaptive systems represent a second category of systems that are characterized by large domains. Microsoft Internet Explorer is an example of such a system. It has 31 configurable options on the security tab. There are around 19 trillion possible configurations for this tab [COH 06] and the system should behave correctly in all these conditions. An emerging trend in embedded systems is the ability to adapt to changes in the environment at runtime. In this case, the set of all possible environment settings represents the set of all configurations under which the system is expected to work.

In the context of the DiVA project [DIV 08], the CAS company develops a customer relationship management system. The requirements for this system describe 23 environmental properties which represents 10^7 possible combinations and as many different environments to which the system is expected to adapt. The variables for configuring the system usually have a finite domain. The challenge for testing is thus to select a minimal set of configurations to test the system.

3.2.3. Grammarware and model transformations

An interesting category of system with a large domain consists of all the systems which input data is modeled with a grammar or a metamodel. The programs which input domain can be described with a grammar are known as grammarware [HEN 05], [KLI 05], and usually have an infinite domain. These applications include parsers, refactoring tools, programs analyzers, etc. For example, Figure 3.2 displays an excerpt of the grammar for the Alloy analyzer [JAC 06]. The first rule states that a specification in Alloy is composed of a number of open and paragraph constructs. Because of the “*” symbol, there can be between 0

and an infinite number of open and paragraph. This means that the input domain for the Alloy analyzer is infinite since there can be an infinite number of specifications that conform to the rules in the grammar.

```
specification ::= [module] open* paragraph*
module ::= "module" name [ "[" ["exactly"] name ("," ["exactly"] num)* "]" ]
open ::= ["private"] "open" name [ "[" ref, "+" "]" ] [ "as" name ]
paragraph ::= factDecl | assertDecl | funDecl | cmdDecl | enumDecl | sigDecl
```

Figure 3.2. *Excerpt from alloy grammar*

Concerning systems which input domain is modeled with a metamodel, we call these systems *model transformations*. They are similar to grammarware programs since their input domain is potentially an infinite set of models that are licit input data for the transformation. What is interesting with grammarware and model transformations is that their input domain is explicitly captured in a finite model that can be leveraged for the definition of test adequacy criteria and to systematically identify a finite set of test data.

Since this lecture focuses on testing model transformations, we provide a detailed example of a metamodel in the following. Figure 3.3 displays a metamodel for a simple class diagram modeling language. This metamodel specifies a class model as being a set of CLASSIFIERS and ASSOCIATIONS. A CLASSIFIER is either a CLASS that can have a parent CLASS, a set of ATTRIBUTES and that can be persistent, or a PRIMITIVE DATATYPE. ATTRIBUTES can be primary, they have a name and a type. ASSOCIATIONS have a name and destination and source CLASS.

All the concepts for this simplified class diagram language are represented by classes in the metamodel. These classes have *properties* that are either attributes of primitive type

(e.g. the name attribute in the ASSOCIATION class) or references to other classes. The references have a role name and a multiplicity. For example, the reference from CLASSMODEL to ASSOCIATION has the role name `association` and a multiplicity `*` which means that a CLASSMODEL contains a set of zero or many ASSOCIATIONS. Constraints to restrict the set of licit class models are captured by references and multiplicities on the references.

Classes and properties are usually not expressive enough to specify all constraints on the structure of the modeling language. The Object Constraint Language (OCL) can be used to add constraints and allow us to build a more precise model of the domain. For example, Figure 3.4 displays additional invariants on the simple class diagram metamodel of Figure 3.3, expressed in OCL. The first one specifies that there must be no cycle in the parent relationship between one CLASS and another, which means that a class cannot inherit from itself or one of its parents.

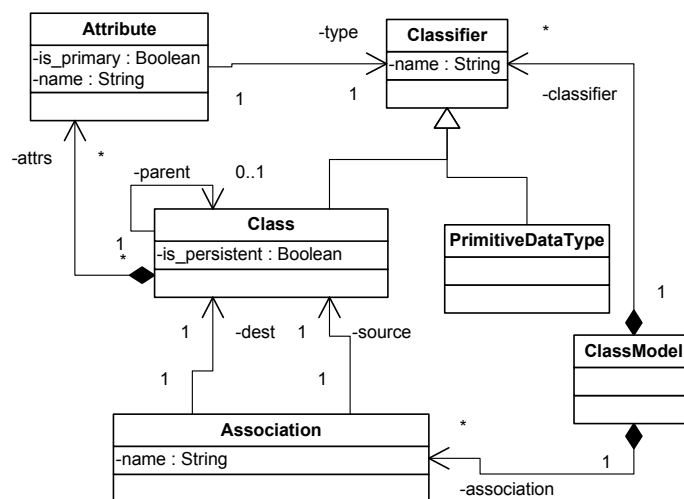


Figure 3.3. Simple UML Class Diagram Metamodel

Invariants on the metamodel**context** Class

```

inv noCyclicInheritance:
    not self.allParents()->includes(self)
inv uniqueAttributeName:
    self.attrs->forAll(att1,att2 | att1.name=att2.name implies att1=att2)

```

context ClassModel

```

inv uniqueClassifierNames:
    self.classifier->forAll(c1,c2 | c1.name=c2.name implies c1=c2)
inv uniqueClassAssociationSourceName:
    self.association -> forAll(ass1,ass2 | ass1.name=ass2.name implies
        (ass1=ass2 or ass1.src!=ass2.src))

```

Figure 3.4. *Additional constraints on the metamodel*

A metamodel defines the input domain of a model transformation. Thus, it defines the set of models that can be passed as input to the transformations. Since the metamodel is defined as a set of classes and properties, a *model is a graph of objects*. The objects in this graph are instances of the classes defined in the metamodel. The structure of the graph is constrained by the multiplicities on references and by all additional constraints defined on the metamodel.

The metamodel of Figure 3.3 models the input domain for any transformation that manipulates simple class diagrams. This metamodel can serve as a basis for the generation of a set of test data. However, the “*” on the cardinality for the set of attributes in a CLASS or the set of associations in a CLASSMODEL, means that there is potentially an infinite number of models that conform to this metamodel. More precisely, the size of the set of classes in a model is between 0 and maxInt. This cardinality alone indicates that the total number of class models that satisfy the structure defined by the metamodel can be very large. The set of classifiers in the model and the set of attributes in a class have the same multiplicity. Thus, the total number of models that combine

only these three properties is maxInt^3 which is $21\,474\,836\,48^3$ for a machine that encodes integers on 32 bits.

3.2.4. Testing challenges

In all the above examples, it appears that, even with small domain models (5 classes in a metamodel or 18 fields in a web form) the number of input data and combinations of data can be very large or even infinite. It is thus impossible to test these systems with all possible data. The issue for test data generation is then to *select a subset of test data in the input domain* according to systematic criteria that cover all relevant sub-domains in that domain.

3.3. Selecting test data in large domains

In this section we introduce category-partition testing [OST 88] and combinatorial testing strategies [COH 96] that can be used separately or conjointly to systematically select a subset of test data in large input domains.

3.3.1. Category partition

The basic idea of category-partition testing strategies [OST 88] is to divide the input domain into sub-domains called *ranges*. This division is based on specific knowledge of the domain and consists of identifying subsets of values that are equivalent with respect to the behavior of the program being tested. The ranges for an input domain define a partition of the input domain and thus should not overlap. Once the partitions and ranges are defined, the test generation consists of selecting one item of test data in each range. *Boundary testing* consists of selecting test data at the boundary of the ranges.

Definition – Partition. *A partition for a variable's domain of elements is a collection of n ranges $R_1, \dots, R_n$ such that $R_1, \dots, R_n$ do not overlap and the union of all subsets forms the domain. These subsets are called ranges.*

In order to use category-partition for test generation, the first step consists of identifying all the variables that define the input domain of the system being tested. These variables can be input parameters for methods, variables that represent the state of the program, environment variables, fields in a form, options to configure the system, or properties in a metamodel. Once these variables are well identified, the domain of each variable must be divided into a set of ranges that form a partition. The process of partition construction is critical: the more representative the values for range boundaries, the more relevant the partitions and thus the more relevant the test data. On the other hand, there are no techniques to automatically identify relevant ranges.

In their introduction to software testing, Amman and Offutt [AMM 08] provide two different approaches for the identification of variables that characterize the input domain and the relevant values for partitioning this domain. The first approach is based on the interface of the system and considers all the variables in isolation. The main benefit of this approach is that it is simple and thus very straightforward to apply, but this might lead to an incomplete domain model (missing links between variables).

The second approach is called the “functionality-based approach”. In this case, the variables are identified according to the expected functionalities of the system. In particular, the variables can be identified in the requirement documents. In this case, the input domain model can be richer and thus more precise.

The construction of partitions for the domains of all variables is the key creative part of this testing approach. The boundaries for ranges in the partition capture representative values for which the system is supposed to behave in a specific way. These values are manually identified. We distinguish between knowledge-based and default partitioning. Knowledge for knowledge-based partitioning can be found:

- in the requirements. For example, the requirements of the form in Figure 3.1 should specify the expected format of phone number or fax numbers. These formats would enable partitioning of the domain of integers for these two variables;
- in the interface design. For example, in Figure 3.1, the variables which domain is captured in a combobox, each value in the combobox is a representative value. Here the ranges in the partition are simple ranges that contain one value each;
- in the pre- and post-conditions of methods if the system is designed by contract. The values in the pre-condition restrict the input domain for an operation. For example, a pre-condition that specifies that an integer parameter should be greater than 5, this indicates that the domain of this variable can be divided in two ranges: greater than 5 and lower than or equal to 5. Similarly post-conditions can provide information on ranges of values that are expected to produce results that satisfy a property;
- in the code itself. Controls on the values of input variables for the system usually capture representative values for these variables. For example, an if a statement can capture a different behavior for the system according to the value of the variable.

Default partitioning can be used when no information is available on representative values. This consists of defining

default ranges to partition the domain for primitive data types. For example, the domain of strings can be partitioned in two ranges: the range that contains the empty string and the range that contains all the non-empty strings. This partition can then guide the selection of String values for the name and organization fields in the form of Figure 3.1. An empty string for these fields is expected to raise an exception since these fields are mandatory in this form.

In model-driven development, partition testing has been adapted to test executable UML models by Andrews *et al.* [AND 03]. They consider a class diagram, OCL pre- and post-conditions [OMG 03] for methods and activity diagrams to specify the behavior of each method. From this model, the authors generate an executable form of the model, which can be tested.

Dinh-Trong *et al.* [DIN 05] then propose to model test cases using UML2.0 sequence diagrams. From these test case specifications and the class diagram, they generate a graph that corresponds to all possible execution paths defined in the different scenarios. The authors then use test criteria defined in [AND 03] and automatically generate test data and an initial system configuration to cover each execution path.

In section 3.4 we show how we adapted category-partition for the definition of test coverage criteria on metamodels [FLE 07].

3.3.2. Combinatorial interaction testing

We have seen how category-partition is a possible technique to reduce infinite domains of variables in a finite set of ranges in which the variables should take at least one value. When all the variables have a finite domain (either using category-partition or because the domain is an

enumeration), there remains one issue for the selection of test input: the selection of a subset of all possible combinations of variables. As we have seen in the example of the web form or of adaptable systems, an important factor for the explosion of the size of the input domain is the number of combinations of variables. In this section we introduce *combinatorial interaction testing* (CIT) [COH 96], [COH 97] as a possible approach to select a subset of all combinations while still guaranteeing a certain level of coverage.

CIT is based on the observation that most of the faults are triggered by interactions between a small number of variables [KUH 04]. This has led to the definition of pairwise testing, or 2-way testing. This technique samples the set of all combinations in such a way that all possible pairs of variable values are included in the set of test data. Pairwise testing has been generalized to t-way testing which samples the input domain to cover all t-way combinations.

For example, let us consider a simple model for a cashier at a movie theater. The variables and the possible values are summarized in Table 3.1. There are 4 types of clients, three periods with different fares, three types of guidance for movies (G for no restriction, PG13 for guidance for children below 13 and R for restriction for children below 17) and three payment methods.

There are 128 possible combinations of values with all the four variables in this simple example. Pairwise testing suggests selecting a subset of all combinations in which all the combinations of pairs of variables are present. A possible solution for pairwise testing with our movie example is displayed in Table 3.1. All the pairs of variables are present, but there are only 12 combinations of input data. This solution is generated by the TConfig tool provided by Alan Williams [WIL 08].

	Client	Period	Parental Guidance	Payment
Possible values	Child	Week	G	Cash
	Adult	Week-end	PG-13	Debit card
	Senior	Holiday	R	Credit card
	Student			

Table 3.1. *Input domain for movie cashier*

Client	Period	Parental Guidance	Payment
Child	Week	G	Cash
Child	Week-end	PG13	Debit card
Child	Holiday	R	Credit Card
Adult	Week	PG13	Credit card
Adult	Week-end	R	Cash
Adult	Holiday	G	Debit card
Senior	Week	R	Debit card
Senior	Week-end	G	Credit card
Senior	Holiday	PG13	Cash
Student	Week	G	Cash
Student	Week-end	PG13	Debit card
Student	Holiday	R	Credit card

Table 3.2. *Pairwise data for movie cashier*

The generation of T-way CIT is based on the generation of a mathematical object called a *covering array*.

Definition – Covering array. A covering array $CA(N; t, k, v)$ is a $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols at least once.

From the definition of a covering array, the strength t of the array is the parameter that enables us to achieve 2-way (pairwise), 3-way or t -way combinations. The k columns on this array correspond to all the variables in the input domain. As it is defined here, all the variables in the array must have the same number v of possible values. Since variables usually do not have the same number of values (e.g. the variables in the movie cashier example have 3 or 4 values), there is a more general structure called a *mixed level covering array*. This array also has k columns, but the variables of each column do not necessarily have the same number of values.

The problem of generating a minimal covering array for a set of variables is a complex optimization problem that has been studied in a large number of works [COH 97], [COH 08], [SHI 04], [WIL 01]. Several tools exist that implement these solutions for CIT automatic generation [CZE 08], [UTT 07a].

It is important to note that there are very few works that have tackled the automatic generation of CIT in the presence of constraints between variables. In our example, there are at least two combinations in Table 3.1 that should raise an exception in the system: the second combination tests the system with a child who sees a PG13 film and the third combination tests it with a child who sees an R film. In order to include properties that forbid combinations of these values, CIT generation techniques have to allow the introduction of constraints in the algorithms that generate a covering array. Recent work by Cohen *et al.* tackles this specific issue [COH 08].

3.4. Metamodel-based test input generation

Models in model-driven development are productive assets for the development of software systems. This means

that models are built in such a way that they can be automatically manipulated by programs. The structure and semantics of models are captured in a metamodel and the programs that manipulate models are referred to as model transformations. The metamodel is thus a model of the input domain for a model transformation.

In Fleurey *et al.* [FLE 07], we have proposed several coverage criteria over a metamodel in order to select and qualify a set of models for testing. These criteria are based on the notion of object and model fragments that define constraints on objects and models that must be present in a set of models adequate for testing. The models that serve as test data for a model transformation are called *test models*.

In this section we introduce how we have adapted category-partition on metamodels to limit the input domain for test models. Then we define the notions of object and model fragments used to define coverage criteria. We also discuss possible strategies to automatically generate models that satisfy these criteria.

3.4.1. Metamodel coverage criteria

In section 3.2.3 we showed that the size of the domain for a model transformation can be very large because of * multiplicities for some properties of the metamodel. In order to restrict the size of the space that has to be explored for test model generation, we define partitions on the domain and/or multiplicity of each property in a metamodel.

Notation – Default partition. *The default partitions for primitive data types are noted as follows:*

– *Boolean partitions are noted as a set of sets of Boolean values. For example, $\{\{false\}\}$ designates a partition with two ranges: a range which contains the value true and a range which contains the value false.*

– *Integer partitions are noted as a set of sets of Integer values. For example, $\{\{0\}, \{1\}, \{x \mid x \geq 2\}\}$ designates a partition with three ranges: 0, 1, greater or equal to 2.*

– *String partitions are noted as a set of sets of String values. A set of string values is specified by a regular expression. For example $\{\{\text{""}\}, \{\text{"+"}\}\}$ designates a partition with two ranges: a range which contains the empty string and a range which contains all strings with one or more character. In the regular expression language, “.” designates any character and “+” specifies that the preceding symbol can be repeated once or more.*

Figure 3.5 displays default partitions and ranges for the simple class diagram metamodel of Figure 3.3 (partitions on the multiplicity of a property are denoted with #). These default partitions, based on the types of properties, are automatically generated for any metamodel by the MMCC tool [MMC 08]. Yet, if there are other representative values in the context of the transformation under test, the tester can enrich the partitions to ensure that they are used in the test models.

Attribute::is_primary	{true}, {false}
Attribute::name	{« »}, {.+}
Attribute::#type	{1}
Class::is_persistent	{true}, {false}
Class::#parent	{0}, {1}
Class::#attrs	{0}, {1}, {x x>1}
Association::name	{« »}, {.+}
Association::#dest	{1}
Association::#source	{1}
ClassModel::#association	{0}, {1}, {x x>1}
ClassModel::#classifier	{0}, {1}, {x x>1}

Figure 3.5. *Partitions for the simple CD metamodel*

Based on this partitioning we can define a simple test adequacy criterion that specifies that each range in each partition should be covered by one test model at least. For

example, the $\{\{\text{""}\}, \{\text{"+"}\}\}$ for the name attribute of ASSOCIATION specifies that there should be one test model that contains an ASSOCIATION with any name that is a non-empty String and another ASSOCIATION that has an empty name. Similarly, the $\{\{0\}, \{1\}, \{x \mid x > 1\}\}$ partition for the multiplicity of the association reference of CLASSMODEL specifies that there should be a test model that has no association, another model that has exactly one association and another model that has more than one association.

Stronger adequacy criteria should require specific combinations of values or ranges of values. A naïve strategy would consist of requiring one test model for each combination of ranges, but even in the simple case of the class diagram language, this would mean generating 1,296 test models. This represents a very large number of models considering the small number of concepts in the metamodel. We thus have defined criteria that limit the number of combinations that have to be covered while ensuring the coverage of the metamodel [FLE 07].

3.4.2. Model and object fragments for test adequacy criteria

A model, an instance of a metamodel, can be seen as a graph of objects that are instances of the classes in the metamodel. The adequacy criteria on a set of test models are defined as constraints on the objects in a test model. We capture the notion of constraint on one object in an *object fragment*. An object fragment constrains the values of certain properties by specifying in which range the property should take its value. It is important to note that an object fragment does not necessarily define constraints for all the properties of a class, but can partially constrain the properties (like a template).

In order to define constraints on the combination of object fragments in complete models, we define the notion of *model fragment*. A model fragment is a collection of object fragments. A model fragment is a constraint that should be satisfied by one test model.

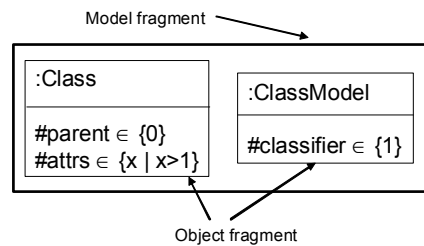


Figure 3.6. Example of object and model fragment

Figure 3.6 displays an example of a model fragment that includes two object fragments. One object fragment [Class::is_persistent {true} Class::#attrs {x | x>1}] specifies that there should be an instance of CLASS in one test model such that the property is_persistent takes its values in the range {true} and the property attrs has a multiplicity in the range {x | x>1}. There is no constraint on the multiplicity of the parents property of the object. The second object fragment specifies that there should be a CLASSMODEL such that the number of classifiers is in the range {1}. There is no constraint on the multiplicity of the association reference of CLASSMODEL. The model fragment specifies that there should be one test model that contains two objects that satisfy both object fragments.

The test adequacy criteria for test models are defined as a set of model fragments that combine ranges of values for the properties according to different strategies. Each criterion specifies a set of model fragments that should be satisfied by a set of test models in order to fulfill the criterion.

Test criterion for metamodel coverage: *A test criterion specifies a set of model fragments for a particular source metamodel. These model fragments are built to guarantee class and range coverage as defined in the following rules.*

Rule 1 - *Class coverage: each concrete class must be instantiated in at least one model fragment.*

Rule 2 - *Range coverage: each range of each partition for all properties of the metamodel must be used in at least one model fragment.*

Test criterion satisfaction for a set of test models: *A set of test models satisfies a test criterion if, for each model fragment MF, there exists a test model M such that all object fragments defined in MF are covered by an object in M. An object O corresponds to an object fragment OF if, for each property constraint in OF, the value for the property in O is included in the range specified by OF.*

The weakest coverage criteria we propose are called *AllRanges* and *AllPartitions*. They both ensure range coverage by combining property constraints in two different manners. *AllRanges* enforces the two rules defined above. *AllPartitions* is a little stronger, as it requires values from all ranges of a property to be used simultaneously in a single test model.

In a metamodel, properties are encapsulated into classes. Based on this structure and on the way metamodels are designed, it is natural that properties of a single class have a stronger semantic relationship with each other than with properties of other classes. To leverage this, we propose four criteria that combine ranges class by class. These criteria differ on the one hand by the number of ranges combinations they require and on the other hand by the way combinations are grouped into model fragments. The formal definition of

the four criteria $\text{Comb}\Sigma$, $\text{Class}\Sigma$, $\text{Comb}\Pi$, $\text{Class}\Pi$ is provided in [FLE 07].

We have built a metamodel [FLE 07] that formally captures the notions of partition for properties in a metamodel, of object and model fragment. This metamodel is the basis for the construction of the MMCC tool [MMC 08]. MMCC can generate partitions for the properties of a metamodel, compute the set of object and model fragments according to an adequacy criterion and check whether a set of models satisfies the criterion.

3.4.3. Discussion

A first important point that has to be noted is that the criteria defined previously are based only on the MOF description of the metamodel. However, the input domain of a transformation is usually modeled with additional constraints. For example, the constraints of Figure 3.4 restrict the set of possible class diagrams. The pre-condition displayed in Figure 3.7 further restricts the input domain of a model transformation with a pre condition on the class diagram metamodel.

Since the definition of model and object fragments does not consider these constraints, some test criteria will require a model fragment in which there is only one class and that this class has only one attribute which is not primary. However, this contradicts the pre-condition of the transformation. Thus, the test criteria might specify uncoverable model fragments. This is a general issue with test adequacy criteria: they define some objectives that cannot be satisfied by any test case. For example, structural test criteria for programs specify infeasible paths [ADR 82], or mutation analysis produces equivalent mutants [OFF 97].

```
pre atLeastOnePrimaryAttribute:
  input.attrs -> select(att1|att1.is_primary)->size()>=1
```

Figure 3.7. *Pre-condition for a transformation on class diagrams*

Another point worth mentioning is the similarities between the coverage criteria on a metamodel and some criteria that have been studied to generate test data for grammarware programs. Amman and Offutt [AMM 08] propose simple criteria to ensure “terminal symbol coverage” and “production coverage” which are very close to the simplest criteria for metamodel coverage: instantiated each metaclass and each association between these classes. Once these minimal criteria are satisfied by data that cover a grammar, more complex criteria consist of combining complex terms to form larger data that test the interactions between rules. In that case, there is the same combinatorial issue as for metamodels. Lämmel *et al.* [LAM 06] directly address this issue and propose “control mechanisms” to limit the explosion. Hennessy *et al.* [HEN 05] study different strategies to limit this explosion and compare them in terms of code coverage and fault detection. In Baudry *et al.* [BAU 05] we proposed a technique driven by mutation analysis in order to limit the generation of test data to data that can kill mutants.

3.4.4. Automatic synthesis of test models

We can expect several benefits from automatic generation of test models. This can save time and effort during the development of a model transformation. This can help when the transformation evolves or the source metamodel changes to take new concepts into account. It can also assist the manual construction of test models with a tool that automatically completes a model to conform to the metamodel.

There are two major challenges for the automatic generation of test models:

- Heterogenous constraints. The constraints that define the input domain and coverage criteria are defined by different actors using different languages. The metamodel is defined by language designers, the restrictions on a metamodel for a specific transformation are defined by transformation developers, test criteria and test objectives are defined by testers. These different models and constraints are expressed with various formalisms: EMOF and OCL for the metamodel, OCL or patterns for the restriction on the input domain, model fragments for test criteria.

- Automatic constraint solving. Adequate test models are defined by a large set of constraints that have to be considered as a whole in order to generate a correct model.

In Sen *et al.* [SEN 07], [SEN 08] we proposed to transform all the different constraints to a common formalism compatible with automatic constraint solving techniques. First we proposed a methodology using *constraint logic programming*. We present a transformation from a metamodel, constraints, fragments and a partial model to a constraint logic program (CLP). We solve/query the CLP to obtain value assignments for undefined properties in the partial model. In a second approach [SEN 08] we proposed to combine all constraints in an Alloy model. Alloy is a lightweight formal modeling language that allows automatic analysis. In particular it is connected to several SAT solvers that can automatically solve a set of constraints and generate instances in the search space.

Other approaches tackle the automatic generation of models to test a model transformation. Two constructive approaches propose generating models first and checking the constraints afterwards. Brottier *et al.* [BRO 06] consider only

the class diagram definition of the source metamodel to generate objects and assemble them according to adequacy criteria based on model fragments in order to build complete models. Ehrig *et al.* [HER 06] analyze the metamodel to generate rules that create instances of all non-abstract classes and links between the instances. The major limitation of these approaches is that they do not consider all the additional constraints on the input domain in the generation process. As a consequence, a large number of generated models do not satisfy the complete set of constraints and thus the transformation cannot process them for testing.

3.5. Conclusion

Automatic model transformations are essential assets in the model-driven development of embedded systems. In this lecture we have focused on testing as a possible approach to assess their quality. In particular we have presented the issues related to the selection of test models in the large space of input models for a transformation.

We have first discussed general issues related to testing systems characterized by large domains. We have introduced category-partition and combinatorial interaction testing (CIT) as two existing techniques to limit the combinatorial explosion test data in the presence of a very large input domain. Then, we have focused on the notion of model and object fragments to define test criteria on a metamodel that models the input domain of a model transformation. These criteria aim at selecting test models in the possibly infinite set of input models for a transformation. We also briefly discussed the challenges for automatic generation of models according to these criteria and possible solution.

There remain many challenges for an efficient selection of test data in large domains. Concerning CIT we mentioned

that it is important to integrate constraints between variables in the generation of testing configurations in order to obtain licit and meaningful combinations. Similarly, the automatic generation of test models must integrate all the constraints on the input domain in order to generate models that can be processed by the transformation. A related issue is the definition of a precise model for the input domain: it is important that all constraints are captured when building this model to allow automatic analysis and effective test generation. Automatic generation of test models also faces usual issues for automatic test generation: interpretation of the test data, management of these data (priorities, regression testing selection, etc.). More generally, there remain important issues for model transformation testing [BAU 09]. The work presented here on selection criteria is a necessary step towards a global solution.

3.6. Acknowledgements

This work has been partially supported by the European project DiVA (EU FP7 STREP). I am also extremely grateful to Franck Fleurey, Jean-Marie Mottu and Sagar Sen whose PhD work has largely contributed to the understanding of model transformation testing as presented here and to Yves Le Traon for the numerous fruitful discussions on software testing.

3.7. Bibliography

- [ADR 82] ADRIO W. R., BRANSTAD M. A. and CHERNIAVSKY J. C..
“Validation, verification, and testing of computer software”,
ACM Computing Surveys, 14 (2), 1982: 159 - 192.
- [AMM 08] AMMANN P. and OFFUTT J., *Introduction to Software Testing*, Cambridge University Press, 2008.

- [ANA 07] ANASTASAKIS K., BORDBAR B. and KÜSTER J. M., “Analysis of model transformations via alloy”, in *Proceedings of MoDeVVa'07 in Conjunction with MODELS'07*, October 2007. Nashville, TN, USA.
- [AND 03] ANDREWS A., FRANCE R., GHOSH S. and CRAIG G.. “Test adequacy criteria for UML design models”, *Software Testing, Verification and Reliability*, 13 (2), 2003: 95 -127.
- [BAU 05] BAUDRY B., FLEUREY F., JÉZÉQUEL J.-M and LE TRAON Y.. “From Genetic to Bacteriological Algorithms for Mutation-Based Testing”, *Software Testing, Verification and Reliability*, 15 (1), 2005: 73-96.
- [BAU 09] BAUDRY B., GHOSH S., FLEUREY F., FRANCE R., LE TRAON Y. and MOTTU J.-M.. “Barriers to systematic model transformation testing”, *Communications of the ACM.*, 2009.
- [BRO 06] BROTTIER E., FLEUREY F., STEEL J., BAUDRY B. and LE TRAON Y., “Metamodel-based test generation for model transformations: an algorithm and a tool”, in *Proceedings of ISSRE'06 (Int. Symposium on Software Reliability Engineering)*. 2006. Raleigh, NC, USA. pp 85 – 94.
- [COH 96] COHEN D. M., DALAL S. R., PARELIUS J. and PATTON G. C.. “The combinatorial design approach to automatic test generation”, *IEEE Software*, 13 (5), 1996: 83-88.
- [COH 97] COHEN D. M., DALAL S. R., FREDMAN M. L. and PATTON G. C. “The AETG system: an approach to testing based on combinatorial design”, *IEEE Transactions on Software Engineering*, 23 (7), 1997: 437-444.
- [COH 06] COHEN M. B., JOSHUA S. and ROTHERMEL G., “Testing across configurations: implications for combinatorial testing”, in *Proceedings of Workshop on Advances in Model-based Software Testing (A-MOST)*, November 2006. Raleigh, North Carolina, USA. pp 1-9.
- [COH 08] COHEN M. B., DWYER M. B. and SHI J.. “Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach”, *IEEE Transactions on Software Engineering*, 34 (5), 2008: 633-650.
- [CZE 08] CZERWONKA J. (2008). Pairwise testing tools, retrieved December, 2008, from <http://www.pairwise.org/tools.asp>.

- [CZE 08] CZERWONKA J. (2008). Pairwise testing tools, retrieved December, 2008, from <http://www.pairwise.org/tools.asp>.
- [DIN 05] DINH-TRONG T., KAWANE N., GHOSH S., FRANCE R. and ANDREWS A., “A tool-supported approach to testing UML design models”, in *Proceedings of ICECCS'05*, June 2005. Shanghai, China. pp 519–528.
- [DIV 08] DiVA. (2008). DiVA EU FP7 STREP, retrieved December, 2008, from <http://www.ict-diva.eu/>.
- [EHR 06] EHRIG K., KÜSTER J. M., TAENTZER G. and WINKELMANN J., “Generating instance models from meta models”, in *Proceedings of FMOODS'06*, June 2006. Bologna, Italy. pp 156–170.
- [FLE 07] FLEUREY F., BAUDRY B., MULLER P.-A. and LE TRAON Y.. “Towards dependable model transformations: Qualifying input test data”, *Software and Systems Modeling*, 2007.
- [GER 00] GÉRARD S., VOROS N. S., KOULAMAS C. and TERRIER F., “Efficient system modeling for complex real-time industrial networks using the ACCORD/UML methodology”, in *Proceedings of DIPES'00*, Paderborn, Germany. pp 11–22, 2000.
- [GOK 08] GOKHALE A., BALASUBRAMANIAN K., BALASUBRAMANIAN J., KRISHNA A., EDWARDS G. T., DENG G., TURKAY E., PARSONS J. and SCHMIDT D. C., “Model driven middleware: a new paradigm for deploying and provisioning distributed real-time and embedded applications”, *Elsevier Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture*, 73 (1), 2008: 39 - 58.
- [HEN 05] HENNESSY M. and POWER J. P., “An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software”, in *Proceedings of ASE'05*, November 2005. Long Beach, CA, USA. pp 104 – 113.
- [JAC 06] JACKSON D., *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2006.
- [KLI 05] KLINT P., LÄMMEL R. and VERHOEF C., “Toward an engineering discipline for grammarware”, *ACM Transactions on Software Engineering and Methodology*, 14 (3), 2005: 331-380.

- [KUH 04] KUHN D. R. and WALLACE D. D.. “Software fault interactions and implications for software testing”, *IEEE Transactions on Software Engineering*, 30 (6), 2004: 418-421.
- [KUS 06] KÜSTER J. M. “Definition and validation of model transformations”, *Software and Systems Modeling*, 5 (3), 2006: 233-259.
- [LAM 06] LÄMMEL R. and SCHULTE W., “Controllable combinatorial coverage in grammar-based testing”, in *Proceedings of TestCom 2006*, May 2006. New York City, USA. pp 19–38.
- [MAD 06] MADL G., ABDELWAHED S. and SCHMIDT D. C., “Verifying distributed real-time properties of embedded systems via graph transformations and model checking”, *Real-time Systems Journal*, 33 (1), 2006: 77-100.
- [MMC 08] MMCC. (2008). Metamodel coverage checker, retrieved December, 2008, from <http://www.irisa.fr/triskell/Softwares/protos/MMCC/>.
- [OFF 97] OFFUTT A. J. and PAN J.. “Automatically Detecting Equivalent Mutants and Infeasible Paths”, *Software Testing, Verification and Reliability*, 7 (3), 1997: 165 - 192.
- [OMG 03] OMG. (2003). UML 2.0 Object Constraint Language (OCL) Final Adopted specification, from <http://www.omg.org/cgi-bin/doc?ptc/2003-10-14>, 2005.
- [OST 88] OSTRAND T. J. and BALCER M. J., “The category-partition method for specifying and generating functional tests”, *Communications of the ACM*, 31 (6), 1988: 676 - 686.
- [SEN 07] SEN S., BAUDRY B. and PRECUP D., “Partial model completion in model driven engineering using constraint logic programming”, in *Proceedings of INAP'07 (International Conference on Applications of Declarative Programming and Knowledge Management)*, October 2007. Würzburg, Germany.
- [SEN 08] SEN S., BAUDRY B. and MOTTU J.-M., “On combining multi-formalism knowledge to select models for model transformation testing”, in *Proceedings of ICST'08 (International Conference on Software Testing Verification and Validation)*, April 2008. Lillehamer, Norway. pp 328-337.

- [SHA 09] SHANKARAN N., KINNEBREW J., KOUTSOUKOS X., LU C., SCHMIDT D. C. and BISWAS G., “An integrated planning and adaptive resource management architecture for distributed real-time embedded systems”, *IEEE Transactions on Computers, Special Issue on Autonomic Network Computing*, 2009.
- [SHI 04] SHIBA T., TSUCHIYA T. and KIKUNO T., “Using artificial life techniques to generate test cases for combinatorial testing”, in *Proceedings of COMPSAC '04: 28th Annual International Computer Software and Applications Conference*, 2004. Washington, DC, USA. pp 72-77.
- [UTT 07a] UTTING M. and LEGEARD B., (2007). MBT Tools, retrieved December, 2008, from <http://www.cs.waikato.ac.nz/~marku/mbt/CommercialMbtTools.pdf>.
- [UTT 07b] UTTING M. and LEGEARD B., *Practical Model-Based Testing*, Morgan Kaufmann, 2007.
- [WIL 01] WILLIAMS A. and PROBERT R. L., “A measure for component interaction test coverage” in *Proceedings of ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, June 2001, Beirut, Lebanon. pp 304-311.
- [WIL 08] WILLIAMS A. (2008). TConfig - Test configuration generator, retrieved December, 2008, from <http://www.site.uottawa.ca/~awilliam/TConfig.jar>.

Chapter 4

Symbolic Execution-Based Techniques for Conformance Testing

4.1. Context

4.1.1. *Conformance testing: an introduction*

Testing is a widely used validation technique to increase software quality. It consists of executing the System Under Test (*SUT*) for some particular use cases and in evaluating whether or not corresponding *SUT*'s executions conform to some requirements. For that, one has to define input test data to be submitted to the *SUT* and a decision procedure (called *oracle*) allowing to assign verdicts depending on *SUT*'s computations. Selecting a set of test data which is both qualitatively and quantitatively sufficient to get confidence in the testing process, is difficult and time consuming. Moreover manually computing test data and assigning verdicts are both long and error prone. Therefore automation is highly desirable. In the framework of model-based development,

Chapter written by Christophe GASTON, Pascale LE GALL, Nicolas RAPIN and Assia TOUIL.

models can help one to automate both test generation and verdict production. We focus on a particular way to do so, which is called *model-based testing* or *black box testing*. Test data are extracted from models, while *SUT* is seen as a black box with which a tester can interact through an interface. In the case of reactive systems, interactions consist in sending inputs and observing *SUT*'s reactions (*i.e.* outputs). Oracles are defined from sets of behaviors specified in models, and so-called *conformance relations* which define the conformity of interactions observed during the testing process with regard to models.

4.1.2. Conformance relation

Labelled transition systems are widely used to specify reactive systems by focusing on the interactions between a system and its environment. These interactions are often modeled as sequences of communication actions, which can be classified as emissions (output messages) or receptions (input messages).

Definition 1. *Let I and O be respectively a set of input labels and a set of output labels. An element of $I \cup O$ is called a communication action.*

An input/output labelled transition system (IOLTS) over (I, O) is a tuple $(Q, q_0, Trans)$ where Q is a set of state names, $q_0 \in Q$ is the initial state and $Trans \subseteq Q \times (I \cup O) \times Q$ is a set of transitions.

A transition (q, l, q') of $Trans$ is often denoted as $q \xrightarrow{l} q'$ and represents the fact that the communication action l allows the system to move from the source state q to the target state q' . In order to easily distinguish output actions from input actions, we conventionally add the symbol ! (resp. ?) behind an output (resp. input) action. Transitions

can be composed to build *paths*. Thus, a finite path is a finite sequence of transitions $(q_0, l_0, q_1)(q_1, l_1, q_2) \dots (q_n, l_n, q_{n+1})$ verifying that any two consecutive transitions are such that the target state of the first one is the source state of the second one. Any such path defines a behavior of the *IOLTS*, also called a *trace*, as the sequence $l_0 l_1 \dots l_n$ of all occurring communication actions. Let us point out that in a black-box testing approach, the observations on a reactive system under test precisely correspond to such finite sequences of input/output communication actions.

Sometimes, in the *IOLTS* used as reference model, there exist states from which the system cannot emit an output message. In such a state, the system can then be perceived as *quiescent*. This situation, i.e. the absence of emission, becomes an observation that the tester can use to check if the *SUT* conforms to the model. Following the approach of [TRE 96b], we introduce a special output communication action, denoted by $\delta!$, expressing that the system cannot emit an output message. We can then enrich the *IOLTS* by adding a new state q_δ and as many transitions $q_i \xrightarrow{\delta!} q_\delta$ as there exist states q_i in which the system is quiescent. Our enrichment by quiescence is a simplified version of the one of [TRE 96b]: in our approach, the $\delta!$ observation can only be observed once in a trace, at its end. On the contrary, in [TRE 96b], the enrichment by quiescence consists of adding loop transitions of the form $q_i \xrightarrow{\delta!} q_i$ for all quiescent states q_i in such a way that quiescence can be repetitively observed, until an input message may allow a state modification. Using a simplified version of the enrichment by quiescence will facilitate the presentation of the test case generation algorithm in section 4.4.3.

The semantics of an *IOLTS* $G = (Q, q_0, Trans)$ is the set $Trace(G)$ of all traces associated with the finite paths of G starting at the initial state q_0 , possibly enriched by quiescence observations. Obviously, the set $Trace(G)$ will serve as a reference to define the conformance relation between a *SUT*

and the model G . Before defining the conformance relation, we have to precisely state which testing hypotheses hold on the SUT . In particular, the SUT should share the same interface as the reference model G , that is the SUT should share the same input and output actions, including the quiescent output. As usual for conformance testing, we consider that the SUT is only observable by its input/output action sequences. Thus, a SUT is naturally defined as a set of traces over its interface. Following our modeling of quiescent states, we suppose that the absence of output from the SUT can be observed through the emission $\delta!$, and in this case, this cannot be directly followed by another emission. In practice, the tester waits for a reaction from the SUT . If the SUT is quiescent during a fixed arbitrary time-out delay, then the tester decides that the SUT cannot definitely emit any output message and thus considers that the SUT emits the $\delta!$ message. Moreover, as the testing activity consists of observing the reaction of the SUT when stimulated with input messages sent by the test case, it means in practice that the SUT should accept any possible input message at any moment. This hypothesis is known under the “input-completeness” hypothesis.

Definition 2 (System under test). *A System Under Test SUT over (I, O) is defined by a set $Trace(SUT)$ of finite sequences over $I \cup O \cup \{\delta!\}$ such that:*

- *$Trace(SUT)$ is stable by prefix: any prefix of a trace in $Trace(SUT)$ also belongs to $Trace(SUT)$,*
- *$Trace(SUT)$ is input-complete: for any $l?$ in I , any trace of $Trace(SUT)$ extended with $l?$ also belongs to $Trace(SUT)$,*
- *any trace of $Trace(SUT)$ terminating by $\delta!$ can be extended only with sequences of input actions.*

Conformance testing assumes that a formal conformance relation is given between the model G and the SUT . Our framework is based on the *ioco* conformance relation which is also used for example in [JAR 04, TRE 96a]. Intuitively a SUT conforms to its model with respect to *ioco* if the reactions of

the *SUT* are the same as those specified when it is stimulated by inputs deduced from the model. This relation has been recognized as well adapted for testing issues. Indeed, models can be under specified insofar as nothing is required for the *SUT* when receiving an input action not specified in the model. Moreover, non-determinism in the model – two transitions are possible in the same state, with different labels or/and different target states - is taken into account by only requiring that the output provided by the *SUT* belongs to the set of all possible non-deterministic outputs of the model.

Definition 3 (ioco conformance relation). *SUT conforms to G if and only if for any $tra \in Trace(G) \cap Trace(SUT)$, if there exists $act \in O \cup \{\delta!\}$ such that $tra.act \in Trace(SUT)$, then $tra.act \in Trace(G)$.*

Model-based testing for *IOLTS* has been intensively studied both from theoretical (e.g. [TRE 96b, TRE 08]) and practical points of view including case studies and tools, (e.g. [BEL 05, JAR 04]). Roughly speaking, since *IOLTS* are particular graphs, test case generation algorithms are based on some specialized graph traversal algorithms: a *FAIL* verdict is computed each time the *SUT* emits an output which is not present in the graph. Moreover, some approaches introduce the notion of test purposes to guide the testing activity towards some behaviors selected by the tester as the behaviors of interest to be tested. An *INCONC* (for *inconclusive*) verdict is then computed when the behavior of the *SUT* meets the requirements of the model but not those of the test purpose. Nevertheless, the main drawback of a model-based testing framework based on *IOLTS* is the lack of expressiveness of *IOLTS* as reference models, in particular with respect to the absence of data types. Indeed, in practice, non-deterministic choices could often be removed by introducing data types. Indeed, triggering of transitions can be restricted by equipping transitions with a guard expressed on some state variables. For example, let us specify

a simple ATM system. When an user asks for an amount (*amount?*), then we would be able to specify that according to some conditions, e.g. the sum available on the user account, the ATM system will satisfy the user request (*cash!*), or on the contrary will send a refusal message (*screen!*). In an *IOLTS* model, the request *amount?* would be followed by a non-deterministic choice between *cash!* and *screen!*. Consequently, there is no way during the testing process to control this non-deterministic choice. In particular, even if the tester considers a test purpose focusing on behaviors including *cash!* messages, then the *SUT* can systematically deviate from the test purpose while remaining correct with respect to the model: it suffices that the *SUT* always emits the *screen!* message in reaction to the request *amount?*. To circumvent this problem, we will use in the following Input Output Symbolic Transition Systems (*IOSTS*) as models: basically, they are *IOLTS* enriched with simple data types.

4.1.3. *An overview of the approach*

IOSTS are composed of a data part and of a state-transition graph part. They specify behaviors of reactive systems with some benefits compared to the use of classic labeled transition systems. Models are often smaller and it is even possible to finitely denote systems having an infinite number of states. Behaviors depend upon states of the specified system. Such states are modeled as assignments of distinguished variables called *attribute variables*. State modifications may be induced by internal operations modeled by attribute variable substitutions, or due to interactions with the environment which are modeled by so called *communication actions* consisting of value exchanges through communication channels. State modifications may be conditioned by first order formulas upon attribute variables and called *guards*.

Approaches based on symbolic transformations make it possible to exploit a particular analysis technique, the

so-called *symbolic execution* [CLA 76, KIN 75], to define a test selection strategy. This technique was first defined to compute program executions according to some constraints expressed on input values. In our contribution, *test purposes* are defined as some particular subtrees of this symbolic execution tree. They may be chosen by the user but we also propose criteria to automatically compute tests purposes. This is a response to industrial needs where engineers are not always able to define which behaviors they want to test. According to these test purposes, test cases are generated. Our algorithm for test case generation is given by a set of inference rules. Each rule is dedicated to handle an observation from the system under test (*SUT*) or a stimulation sent by the test case to the *SUT*. This testing process leads to a verdict being either *PASS*, *FAIL*, *INCONC* or *WeakPASS*. *PASS* means that the *SUT* succeeded in passing a test. *FAIL* means that a non-conformance has been detected. *INCONC* means that conformance is observed but the test purpose is not achieved while *WeakPASS* means that we are not sure if we have achieved the test purpose. This last case is essentially due to the fact that the models may be non-deterministic.

4.2. Input output symbolic transition systems

4.2.1. Data types

Data types are specified with a *typed equational* specification framework.

4.2.1.1. Syntax

A data type signature is a couple $\Omega = (S, Op)$ where S is a set of type names, Op is a set of operation names, each one provided with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ (for $i \leq n$, $s_i \in S$). Let $V = \bigcup_{s \in S} V_s$ be a set of typed variable names. The set of Ω -terms with variables in V is denoted $T_\Omega(V) = \bigcup_{s \in S} T_\Omega(V)_s$ and

is inductively defined as usual over Op and V . $T_\Omega(\emptyset)$ is simply denoted T_Ω .

An Ω -substitution is a function $\sigma : V \rightarrow T_\Omega(V)$ preserving types. In the following, we note $T_\Omega(V)^V$ the set of all the Ω -substitutions of the variables V . Any substitution σ may be canonically extended to terms.

The set $Sen_\Omega(V)$ of all typed equational Ω -formulae contains the truth values *true*, *false* and all formulae built using the equality predicates $t = t'$ for $t, t' \in T_\Omega(V)_s$, and the usual connectives $\neg, \vee, \wedge, \Rightarrow$.

4.2.1.2. Semantics

A Ω -model is a family $M = \{M_s\}_{s \in S}$ with, for each $f : s_1 \cdots s_n \rightarrow s \in Op$, a function $f_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow M_s$. We define Ω -interpretations as applications ν from V to M preserving types, extended to terms in $T_\Omega(V)$. A model M satisfies a formula φ , denoted by $M \models \varphi$, if and only if, for all interpretations ν , $M \models_\nu \varphi$, where $M \models_\nu t = t'$ is defined by $\nu(t) = \nu(t')$, and where the truth values and the connectives are handled as usual. M^V is the set of all Ω -interpretations of V in M . Given a model M and a formula φ , φ is said to be *satisfiable* in M , if there exists an interpretation ν such that $M \models_\nu \varphi$.

In the following, we assume that data types of *IOSTS* correspond to the generic signature $\Omega = (S, Op)$ and are interpreted in a fixed model M . In the following, elements of M are called *concrete data* and denoted by terms of T_Ω .

4.2.2. Input/output symbolic transition systems

Definition 4 (IOSTS-signature). An *IOSTS-signature* Σ is a triple (Ω, A, C) where Ω is a data type signature, $A = \bigcup_{s \in S} A_s$ is a set of variable names called *attribute variables* and C is a set of names whose elements are called *communication channels*.

An *IOSTS* communicates with its environment by means of *communication actions*:

Definition 5 (Actions). *The set of communication actions, denoted $Act(\Sigma) = Input(\Sigma) \cup Output(\Sigma)$, is defined as follows, with $c \in C$, $y \in A$ and $t \in T_\Omega(A)$:*

$$Input(\Sigma) = c?y \mid c? \quad \text{and} \quad Output(\Sigma) = c!t \mid c!$$

Elements of $Input(\Sigma)$ are stimulations of the system from the environment: $c?x$ (resp. $c?$) means that the system waits on the channel c for a value that will be assigned to the attribute variable x (resp. for a signal, for example, a pressed button). $Output(\Sigma)$ are responses of the system to the environment: $c!t$ (resp. $c!$) is the emission of the value t (resp. of a message without any sensible argument) through the channel c .

Definition 6 (IOSTS). *An *IOSTS* over Σ is a triple $G = (Q, q_0, Trans)$ where Q is a set of state names, $q_0 \in Q$ is the initial state and $Trans \subseteq Q \times Act_\Sigma(A) \times Sen_\Omega(A) \times T_\Omega(A)^A \times Q$. A transition (q, act, ϕ, ρ, q') of $Trans$ is composed of a source state q , an action act , a guard ϕ , a substitution of variables ρ and a target state q' . For each state $q \in Q$, there is a finite number of transitions of source state q .*

In the frame of *IOSTS*, quiescence from q depends on the current values of the attribute variables and on the guards of all transitions outgoing from q . As explained in section 4.1, we can complete an *IOSTS* to explain quiescent situations. For that, we add a special output communication action $\delta!$, expressing the absence of output, whose guard is complementary to all other guards of output transitions from q .

Definition 7 (Enrichment by quiescence). *Let $G = (Q, q_0, Trans)$ be an *IOSTS* over $\Sigma = (\Omega, A, C)$. The enrichment of G by quiescence is the *IOSTS* over $\Sigma_\delta = (\Omega, A, C \cup \{\delta\})$, defined*

by $G_\delta = (Q \cup \{q_\delta\}, q_0, Trans \cup Trans_\delta)$ where $(q, act, \varphi, \rho, q') \in Trans_\delta$ if and only if:

- $act = \delta!$, ρ is the identity substitution and $q' = q_\delta$,
- Let us note $tr_1, \dots, tr_n$ all transitions of the form $tr_i = (q, act_i, \varphi_i, \rho_i, q_i)$ with $act_i \in Output(\Sigma)$. Then φ is $\bigwedge_{i \leq n} \neg \varphi_i$ if $n > 0$ and is true otherwise¹.

Example 1. Let us consider an ATM system built over the communicating automaton depicted in Figure 4.1. This IOSTS specifies a system of cash withdrawal, with the initial state q_0 . The user asks for some amount (*amount?x*). The ATM system checks if there is enough money in the account user (represented by the variable m) and if this is the first or the second time that the user withdraws money after a deposit. Then the user receives the asked amount by the channel *cash*. If the user account is less than 1000 then the withdrawal operation is not free and costs 1. Otherwise, if there is not enough money in the account, the user receives an error message by the channel *screen*. The user can also deposit some money (t) in his bank account by the channel *deposit*. This is added to the bank account ($m := m + t$). Moreover, the user can ask for the amount of its account by the channel *check*, and receives the answer by the channel *sum*. There is only one transition labeled by $\delta!$ starting from the state q_0 . Indeed, the states q_1 and q_2 are such that whatever the values of the attribute variables are, it is always possible to emit at least one message.

4.2.3. Semantics

The basic notion used to define semantics of IOSTS represents atomic executions of transitions of IOSTS. We call such executions *runs of transitions*. This consists of

1. If $\bigwedge_{i \leq n} \neg \varphi_i$ is not a satisfiable formula, the $(q, act, \bigwedge_{i \leq n} \neg \varphi_i, \rho, q')$ transition may clearly be omitted.

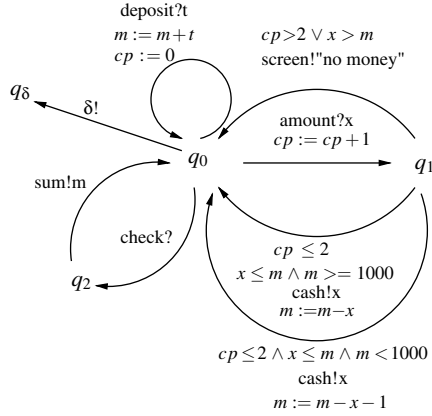


Figure 4.1. Example of ATM system with withdrawal according to some conditions

two interpretations of attribute variables, one before the execution and one after, together with a message obtained by assigning a proper value to the term occurring in the communication action (such messages belong to the set $Act(M)$ in Definition 8).

Definition 8 (Runs of a transition). Let $tr = (q, act, \varphi, \rho, q') \in Trans$. Let us note $Act(M) = (C \times \{?, !\} \times M) \cup (C \times \{?, !\})$. The set $Run(tr) \subseteq M^A \times Act(M) \times M^A$ of execution runs of tr is such that $(\nu^i, act_M, \nu^f) \in Run(tr)$ if and only if:

– if act is of the form $c!t$ (resp. $c!$) then $M \models_{\nu^i} \varphi$, $\nu^f = \nu^i \circ \rho$ and $act_M = (c, !, \nu^i(t))$ (resp. $act_M = (c, !)$),

– if act is of the form $c?y$ then $M \models_{\nu^i} \varphi$, there exists ν^a such that $\nu^a(z) = \nu^i(z)$ for all $z \neq y$, $\nu^f = \nu^a \circ \rho$ and $act_M = (c, ?, \nu^a(y))$,

– if act is of the form $c?$ then $M \models_{\nu^i} \varphi$, $\nu^f = \nu^i \circ \rho$ and $act_M = (c, ?)$.

We denote $source(tr)$ (resp. $target(tr)$) the source (resp. target) state q (resp. q') and $act(tr)$ stands for act . For a run

$r = (\nu^i, \text{act}_M, \nu^f)$, we denote $\text{source}(r)$, $\text{act}(r)$ and $\text{target}(r)$ respectively ν^i , act_M and ν^f .

Executions characterized by an *IOSTS* are modeled as sequences of runs of consecutive transitions. Such sequences are called *finite paths*.

Definition 9 (Finite Paths of an *IOSTS*). *The set of finite paths in G , denoted $FP(G)$ contains all finite sequence $tr_1 \dots tr_n$ of transitions in $Trans$ such that $\text{source}(tr_1) = q_0$ and for all $i < n$, $\text{target}(tr_i) = \text{source}(tr_{i+1})$.*

The runs of a finite path $tr_1 \dots tr_n$ in $FP(G)$ are sequences $r_1 \dots r_n$ such that for all $i \leq n$, r_i is a run of tr_i and for all $i < n$, $\text{target}(r_i) = \text{source}(r_{i+1})$.

The set of concrete traces of a finite path $p = tr_1 \dots tr_n$, denoted $\text{Trace}(p)$ is the set of finite action sequences $\text{act}(r_1) \dots \text{act}(r_n)$ for any run $r_1 \dots r_n$ of p .

Assigning a meaning to *IOSTS*, and intermediately to finite paths, is made for testing issues. Since we propose an approach in the context of black box testing, *SUT* is only observable by means of its reactions to stimulations. This motivates the introduction of the notion of traces of finite paths in Definition 9, and the way we use it to define semantics of *IOSTS*s.

Definition 10 (*IOSTS* semantics). *The semantics of an *IOSTS* G is $\text{Trace}(G) = \bigcup_{p \in FP(G)} \text{Trace}(p)$*

4.3. Symbolic execution

In our context, we call a *symbolic behavior* of an *IOSTS* any finite path p of this *IOSTS* for which $\text{Trace}(p) \neq \emptyset$. In order to characterize the set of traces of a symbolic behavior we propose to use a *symbolic execution* mechanism. Symbolic execution

was first defined for programs [KIN 75, CLA 76, RAM 76]. This technique can naturally be adapted to the framework of *IOSTS*. The main idea is to replace concrete input values and initialization values of attribute variables with symbolic ones with fresh variables and to compute the constraints on these variables: these constraints are called *path conditions*. In the following we assume that those fresh variables are chosen in a set $F = \bigcup_{s \in S} F_s$ disjoint from the set of attribute variables A . We

now give the intermediate definition of *symbolic extended state* which is a structure allowing us to store information about a symbolic behavior: the *IOSTS* current state (target state of the last transition of the symbolic behavior), the path condition and the symbolic values associated with attribute variables.

Definition 11 (Symbolic extended state). A symbolic extended state over F for an *IOSTS* $G = (Q, q_0, Trans)$ is a triple $\eta = (q, \pi, \sigma)$ where $q \in Q$, $\pi \in Sen_\Omega(F)$ is called a path condition and $\sigma \in T_\Omega(F)^A$. $\eta = (q, \pi, \sigma)$ is said to be satisfiable if π is satisfiable². We note S (resp. S_{sat}) the set of all the (resp. satisfiable) symbolic extended states over F .

We now define the symbolic execution of an *IOSTS*. Intuitively, the symbolic execution of an *IOSTS* can be seen as a tree whose edges are symbolic extended states and vertices are labeled by symbolic communication actions. The root is a symbolic extended state made of the *IOSTS* initial state, the path condition *true* (there is no constraint to begin the execution) and of an arbitrary initialization σ_0 of variables of A in F . Vertices are computed by choosing a source symbolic state η already computed and by symbolically executing a transition of the *IOSTS* whose source is the state introduced in η . The symbolic communication action is computed from the transition communication action and from the symbolic values

2. Let us recall that here, π is *satisfiable* if and only if there exists $\nu \in M^F$ such that $M \models_\nu \pi$ since variables of π are by construction in F .

associated with attribute variables in η . A target symbolic extended state is then computed. It stores the target state of the transition, a new path condition derived from the path condition of η and from the transition guard, and finally the new symbolic values associated to attribute variables.

Definition 12 (Symbolic execution of an *IOSTS*). *Let $G = (Q, q_0, Trans)$ be an *IOSTS* over $\Sigma = (\Omega, A, C)$. Let us note $\Sigma_F = (\Omega, F, C)$. A full symbolic execution of G over F is a triple $(S, init, R)$ with $init = (q_0, true, \sigma_0)$ where σ_0 is an injective substitution in F^A and $R \subseteq S \times Act(\Sigma_F) \times S$ such that for any two transitions in R respectively of the form $(\eta^i, c?x, \eta^f)$ and $(\eta^i, d?y, \eta^f)$, the variables x and y are distinct and $\forall a \in A, \sigma_0(a) \neq x$. For any $\eta \in S$ of the form (q, π, σ) , for all $tr \in Trans$ of the form $(q, act, \varphi, \rho, q')$, there exists a unique symbolic transition $st = (\eta, sa, \eta')$ in R such that:*

- if $act = c!t$ (resp. $c!$), then $sa = c!\sigma(t)$ (resp. $c!$) and $\eta' = (q', \pi \wedge \sigma(\varphi), \sigma \circ \rho)$,
- if $act = c?x$ with x in A then $sa = c?z$ with z in F , and $\eta' = (q', \pi \wedge \sigma(\varphi), \sigma \circ (x \mapsto z) \circ \rho)$,
- if $act = c?$ then $sa = c?$, and $\eta' = (q', \pi \wedge \sigma(\varphi), \sigma \circ \rho)$.

The symbolic execution of G over F is the triple $SE(G) = (S_{sat}, init, R_{sat})$ where R_{sat} is the restriction of R to $S_{sat} \times Act(\Sigma_F) \times S_{sat}$.

The trace semantics for a symbolic execution tree is defined in a natural way. If we solves the path condition of a given path (i.e. the path condition of its last state) we can then evaluate all symbolic actions labeling this path and extract the corresponding trace. Since $SE(G)$ is obtained from the symbolic execution tree of G by removing only un-solvable paths, we can easily prove that $Trace(G) = Trace(SE(G))$. Finally, since an *IOSTS* and its symbolic execution share the same trace semantics, it is equivalent to study an *IOSTS* or its symbolic execution in the context of conformance testing.

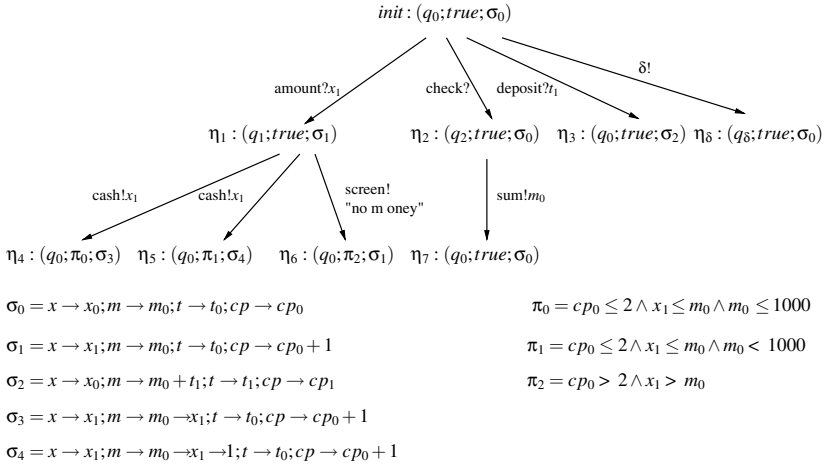


Figure 4.2. Symbolic execution tree

Figure 4.2 illustrates the beginning of the symbolic execution of the ATM system presented in Figure 4.1. At the initialization step, the system is in the state q_0 provided with the initial condition *true*. Now, the ATM system can evolve if the system receives a value for the variable x from the environment through the channel *amount*, or if it receives a value for the variable t from the environment through the channel *deposit* or if the channel *check* is stimulated (with no argument) or if δ occurs (time out). It corresponds to four symbolic transitions yielding to new symbolic extended states: η_1 , η_2 , η_3 and η_δ . The construction of the symbolic execution tree is pursued in the same way.

4.4. Conformance testing for *IOSTS*

In this section we present the symbolic model-based testing framework introduced in [GAS 06]. Following section 4.1, that framework is based on the *ioco* conformance relation presented in Definition 3. The *SUT* is a set of traces as defined in Definition 2. It is defined over the interface (I, O) with $I =$

$(C \times \{?\} \times M) \cup (C \times \{?\})$ and $O = (C \times \{!\} \times M) \cup (C \times \{!\})$. Thus we have $Act(M) = I \cup O$.

4.4.1. Test purposes

Test purposes are used to select some behaviors to be tested. In our case, test purposes consist of some finite paths of the symbolic execution of the model. For each of those paths, the last symbolic extended state is the target state of an output action and is labeled by the keyword *accept*. Restriction on the last actions being necessarily output actions is imposed because our algorithm produces verdicts only with respect to reactions of *SUT*. All states belonging to a chosen path (except the last one labeled *accept*) are labeled *skip*. A *skip* label simply means that it is still possible to reach an *accept* state by emitting or receiving additional messages. So, a test purpose is a finite subtree of the symbolic execution whose leaves are labeled by *accept* and intermediate nodes are labeled by *skip*. All other states, external to the test purpose, are labeled by $\odot$: they are not meaningful with respect to the selected paths of the test purpose.

Definition 13. Let G be an IOSTS with $SE(G_\delta) = (\mathcal{S}_{sat}, init, R_{sat})$ its associated symbolic execution. A symbolic test purpose for G is an application $TP : \mathcal{S}_{sat} \rightarrow \{skip, accept, \odot\}$ such that:

- there exists η verifying $TP(\eta) = accept$,
- for any η, η' verifying $TP(\eta) = TP(\eta') = accept$, there is no finite path $st_1 \cdots st_n$ such that for some $i \leq n$, $source(st_i) = \eta$ and $target(st_n) = \eta'$,
- for any η' verifying $TP(\eta') = accept$, there exists (η, sa, η') in $SE(G_\delta)$ such that sa is of the form $c!t$ or $c!$.
- $TP(\eta) = skip$ if and only if there exists a finite path $st_1 \cdots st_n$ such that for some $i \leq n$, $source(st_i) = \eta$ and $TP(target(st_n)) = accept$. Otherwise $TP(\eta) = \odot$.

4.4.2. Preliminary definitions and informal description

A test execution consists of executing a transition system on the *SUT*, called a test case which is devoted to producing testing verdicts such as *PASS* or *FAIL*. The test case and the *SUT* share the same set of channels and are synchronized by coupling emissions and receptions on a given communication channel. We focus on the sequence of data exchanged between the test case and the *SUT*. These data are in fact elements of M (the model of the data part) and will be denoted by ground terms of T_Ω . We use the following notations: $obs(c!t)$ with t in T_Ω to characterize that the *SUT* emits through the channel c the concrete value denoted t and $stim(c?t)$ to represent stimulations of the *SUT*, occurring when the data t is sent by the test case to the *SUT*. We also use the following generic notation $[ev_1, ev_2, \dots, ev_n | verdict]$ for a sequence of synchronized transitions between a test case and the *SUT* leading to the verdict *verdict*, each action ev_i being issued either from an observation $obs(ev_i)$ or a stimulation $stim(ev_i)$.

Testing a *SUT* with respect to a given symbolic test purpose amounts to looking for, stimulating and observing the *SUT* in such a way that when conformity is not violated, the sequence of stimulations and observations corresponds to a trace (belonging to semantics) of at least one path of the test purpose.

To reach this goal, the testing process achieves two tasks:

- The first task consists of computing, each time it is required, a stimulation compatible with reaching an *accept* state.
- The second task consists of computing all the symbolic states which may have been reached taking into account the whole sequence of observations/stimulations already encountered. Such an interaction sequence corresponds intuitively to a trace. Potentially reached symbolic extended

states are those of the last states of a symbolic path admitting the interaction sequence as trace.

The notion of *context* is used to store such possibly reached symbolic states, together with formulae reflecting constraints on symbolic variables induced by the interaction sequence.

Definition 14 (Context). *A context is a couple (s, f) where $s \in \mathcal{S}_{sat}$ and f is a formula whose variables are in F .*

As previously pointed out, there may be more than one single context compatible with an interaction sequence. We take into account this point by considering sets of contexts, generically noted SC (for Set of Contexts) in the following, and representing the set of all potential appropriate contexts for a given interaction sequence. We introduce some auxiliary functions useful to reason about sets of contexts, in particular in order to be able to compute the sequence of sets of contexts resulting from the successive application of elementary actions.

Definition 15 (Function $Next(ev, SC)$). *Let SC be a finite set of contexts and $ev \in Act(\Sigma_F)$. If ev is of the form $c\Delta t$ (resp. $c\Delta$) with $\Delta \in \{?, !\}$ then $(s', f') \in Next(ev, SC)$ with $s' = (q', \pi', \sigma')$ if and only if:*

- *there exists $(s, f) \in SC$ such that $(s, c\Delta u, s') \in R$ (resp. $(s, c\Delta, s') \in R$),*
- *f' is $f \wedge (t = u)$ (resp. f) and $f' \wedge \pi'$ is satisfiable.*

$Next(ev, SC)$ contains contexts (s', f') built over symbolic states s' reachable from at least one symbolic extended state s structured in a context (s, f) of SC . The constraint f' is f when ev is simply a signal, and otherwise is the conjunction of the previously computed constraint f together with the equality $(t = u)$ which identifies the term u occurring in the

symbolic action of the transition linking s and s' with the term t occurring in ev .

When stimulating the SUT , it is important to check whether the computation of a stimulation is compatible with the goal of finally reaching an *accept* state. For that, for any context ct , the $targetCond(ct)$ predicate allows us to confront constraints inherited from the first observations or stimulations to the target states labeled by *accept* in the test purpose.

Definition 16 ($targetCond(ct)$). Let $ct = (s, f)$ be a context such that $TP(s) = skip$ and³ $E = \{s' \in \mathcal{S}_{sat} \mid \exists m \in (Act(\Sigma_F))^*, s \xrightarrow{m} s' \text{ and } TP(s') = accept\}$, then $targetCond(ct)$ is the formula: $\bigvee_{(q,\pi,\sigma) \in E} \pi$.

The predicate $targetCond(ct)$ characterizes the condition that has to be true so that it is still possible to interact with SUT to reach a state labeled by *accept*. To compute that condition, we identify the set E of symbolic extended states labeled by *accept* and reachable (consistently with the transition relation of the symbolic execution) from the symbolic state structured in ct . In order to reach a state labeled by *accept*, at least one of the path conditions of a symbolic extended state in E must be satisfied. Therefore $targetCond(ct)$ is the formula: $\bigvee_{(q,\pi,\sigma) \in E} \pi$.

Given a set of contexts SC , we distinguish among all contexts in $Next(ev, SC)$ those which are pertinent with respect to the considered test purpose:

3. For a labeled graph G and a word $m = a_1 \cdot \dots \cdot a_n$, the notation $s_0 \xrightarrow{m} s_n$ stands for any path $s_0 \xrightarrow{a_1} s_1 \dots s_{n-1} \xrightarrow{a_n} s_n$ where each $s_i \xrightarrow{a_i} s_{i+1}$ is a transition of G .

Definition 17 (Functions $NextSkip(ev, SC)$ and $NextPass(ev, SC)$). Let SC be a finite set of contexts and $ev \in Act(\Sigma_F)$. If ev is of the form $c\Delta t$ (resp. $c\Delta$) with $\Delta \in \{?, !\}$ then $(s', f') \in NextSkip(ev, SC)$ if and only if:

- there exists $(s, f) \in SC$ such that $(s, c\Delta u, s') \in R$ (resp. $(s, c\Delta, s') \in R$) with $TP(s') = skip$,
- f' is $f \wedge (t = u)$ (resp. f) and $f' \wedge targetCond(s')$ is satisfiable.

$NextPass(ev, SC)$ is defined in the same way with the difference that $TP(s')$ is required to be *accept* instead of *skip*.

Let us remark that for a given symbolic state $s' = (q', \pi', \sigma')$, the predicate $targetCond(s')$ is necessarily stronger⁴ than π' since by definition of symbolic execution, the set of constraints is increasing at each new transition. Thus, we get $NextSkip(ev, SC) \subseteq Next(ev, SC)$ and $NextPass(ev, SC) \subseteq Next(ev, SC)$ for all contexts SC and events ev . $NextSkip(ev, SC)$ is the subset of $Next(ev, SC)$ of contexts, for which it is still possible to build an interaction sequence leading to a symbolic extended state labeled by *accept*. That is ensured by both, the fact that symbolic extended states occurring in contexts of $NextSkip(ev, SC)$ are labeled by *skip* and the conjunction of their constraints with their associated $targetCond$ predicate is satisfiable. Emptiness of $NextSkip(ev, SC)$ means that *accept* is now no longer reachable. In the same way, $NextPass(ev, SC)$ is the subset of $Next(ev, SC)$ of contexts for which a symbolic extended state labeled by *accept* has been reached. That is ensured by both, the fact that symbolic extended states occurring in contexts of $NextPass(ev, SC)$ are labeled by *accept* and the conjunction of their constraints with their associated $targetCond$ predicate is

4. π' is said to be stronger than π if and only if for any interpretation ν , if $M \models_\nu \pi'$, then $M \models_\nu \pi$.

satisfiable. Non-emptiness of $NextPass(ev, SC)$ means that at least an *accept* has been reached.

Let us illustrate our algorithm with Figure 4.3 and describe an execution step based on an emission ev from the *SUT* and starting from $SC = \{(\eta_0, \varphi_0), (\eta_1, \varphi_1)\}$. If $Next(ev, SC)$ is empty, that is the case for $ev = e!x$, this means that the emission is not specified and so we conclude *FAIL* (see Figure 4.3 (4)). If an *accept* is reached ($NextPass(ev, SC)$ non-empty) we conclude *PASS* when no other context is reached, see for example Figure 4.3 (1) with $NextPass(c!t, SC) = \{(\eta_2, \varphi_2)\}$, or *WeakPASS* when others contexts are also reached, see for example Figure 4.3 (3) with $Next(c!t, SC) = \{(\eta_2, \varphi_2), (\eta_3, \varphi_3)\}$. In this last case, we cannot distinguish whether the inner state of the *SUT* is represented by the reached *accept* state (η_2, φ_2) or by the state (η_3, φ_3) outside of the test purpose. At last, if $NextSkip(ev, SC)$ is empty while $Next(ev, SC)$ is not, see Figure 4.3 (2) for $ev = d!l$, this means that the emission was specified but was not aimed by the

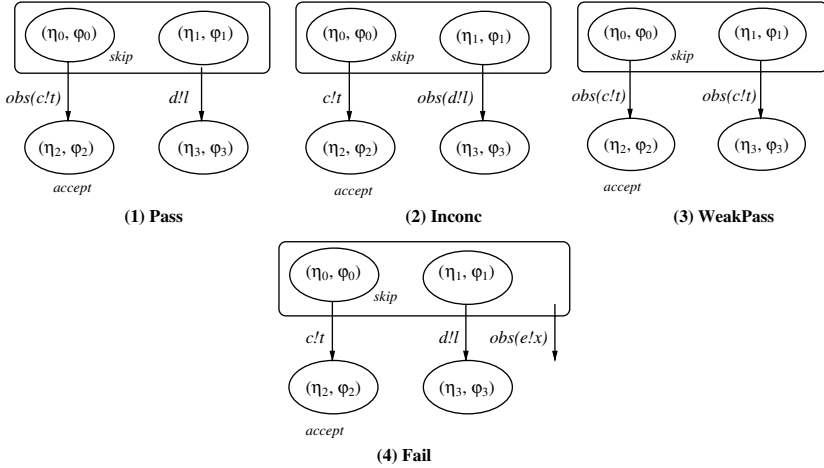


Figure 4.3. Algorithm's explanations

test purpose. Then, we conclude by an inconclusive verdict *INCONC*.

4.4.3. Inference rules

Let us recall that our goal is to compute sequences $[ev_1, \dots, ev_n | Verdict]$ representing synchronized transitions between a test case and the *SUT* leading to the verdict *Verdict*, each action ev_i being derived either from an observation $obs(ev_i)$ or a stimulation $stim(ev_i)$, and *Verdict* belonging to this set of keywords: $\{PASS, WeakPASS, INCONC, FAIL\}$. For that, we will take into account the knowledge of the associated contexts. Each step of the construction of such a sequence will be described by means of inference rules. Those rules are structured as follows⁵ $\frac{SC}{Result} \text{ cond}(ev)$ where *SC* is a set of contexts, *Result* is either a set of contexts or a verdict, $cond(ev)$ is a set of conditions including the observation $obs(ev)$ or the stimulation $stim(ev)$. One should read a rule as follows: *Given the current set of contexts SC, if cond(ev) is verified then the algorithm may achieve a step of execution, with ev as elementary action.* As long as *Result* is a set of contexts, a new rule may be applied to pursue the computation of the sequence. Of course, reaching a verdict stops the algorithm.

Rule 0: Initialization rule.

$$\overline{\{(init, true)\}}$$

Rule 1: The emission is compatible with the purpose but no *accept* is reached.

$$\frac{SC}{Next(ev, SC)} \quad obs(ev), NextSkip(ev, SC) \neq \emptyset, NextPass(ev, SC) = \emptyset$$

5. The initialization rule will not respect this generic structure since it will simply consist in introducing the starting context.

Rule 2: The emission is not expected with regards to the model.

$$\frac{SC}{FAIL} \text{ obs}(ev), \text{Next}(ev, SC) = \emptyset$$

Rule 3: The emission is specified but not compatible with the test purpose.

$$\frac{SC}{INCONC} \text{ obs}(ev), \text{Next}(ev, SC) \neq \emptyset, \text{NextSkip}(ev, SC) = \emptyset, \\ \text{NextPass}(ev, SC) = \emptyset$$

Rule 4: All the next contexts are *accept* ones.

$$\frac{SC}{PASS} \text{ obs}(ev), \text{Next}(ev, SC) = \text{NextPass}(ev, SC), \text{Next}(ev, SC) \neq \emptyset$$

Rule 5: Some of the next contexts are labeled by *accept*, but not all of them.

$$\frac{SC}{WeakPASS} \text{ obs}(ev), \text{NextPass}(ev, SC) \neq \emptyset, \\ \text{NextPass}(ev, SC) \subsetneq \text{Next}(ev, SC)$$

Rule 6: Stimulation of the *SUT*

$$\frac{SC}{\text{Next}(ev, SC)} \text{ stim}(ev), \text{NextSkip}(ev, SC) \neq \emptyset$$

Rules from 1 to 5 concern observations while only Rule 6 concerns stimulations. Rule 5 calls for some comments: a verdict *WeakPASS* means both that the test purpose is reached and that the sequence of observations/stimulations may correspond to another behavior of the symbolic execution. This verdict is thus a kind of *warning*. One should pursue the test execution sequence to distinguish which states really correspond to the performed execution sequence.

We note $st(TP, SUT)$ the set of $[ev_1, \dots, ev_n | \text{Verdict}]$ such that $ev_1 \dots ev_n$ is a sequence of synchronized transitions between $TS(TP)$ and SUT leading to the final state labeled

by *Verdict* in $TS(TP)$. Finally, we introduce the notation:

$$vdt(TP, SUT) = \{Verdict \mid \exists ev_1, \dots, ev_n, [ev_1, \dots, ev_n \mid Verdict] \in st(TP, SUT)\}$$

Using these notations, we can now state the correctness and the completeness of our algorithm:

Theorem 1. *For any IOSTS G and any SUT :*

Correctness: *If SUT conforms to G , for any symbolic test purpose TP ,
 $FAIL \notin vdt(TP, SUT)$.*

Completeness: *If SUT does not conform to G , there exists a symbolic test purpose TP such that $FAIL \in vdt(TP, SUT)$.*

The completeness property holds up to all the non-deterministic choices induced by our set of rules and captured in the set $vdt(TP, SUT)$.

4.5. Concluding remarks

4.5.1. Choosing test purposes

IOSTS models are used to describe reactive systems that continuously interact with their environment. Therefore, such models characterize sets of traces that are arbitrary long. The impact on symbolic treatments of such models is that their associated symbolic trees are generally infinite. Each arbitrary long path of such trees is *a priori* a test purpose to be considered. It is of course impossible to deal with an infinite number of test purposes in a testing process. A usual approach consists of basing test purpose choices on some expert knowledge. These experts are supposed to manually define test purposes corresponding to interesting behaviors to be tested. In our approach this can be done by choosing manually some paths of a previously computed bounded symbolic execution as test purposes. However some more subtle approaches exist. In [JAR 04, JEA 05] test purposes

corresponding to properties to be tested are identified by an expert before any computation based on the model. In those two works, test purposes and models are encoded in the form automata: *IOLTS* in [JAR 04] and *IOSTS* in [JEA 05]. Test purposes and models share the same set of channels. In both cases those test purposes are used to extract test cases from the model. Extraction is done by realizing a synchronous product between the test purpose and the model. This operation results in the definition of an automaton (*IOLTS* or *IOSTS*), called a *test case*, some particular states of which are labeled by a verdict of the form *PASS*, *FAIL* or *INCONC*. Intuitively, the meaning of those verdicts is the same as the one we gave to our *PASS*, *FAIL* or *INCONC* verdicts. Each path of the test case characterizes a set of traces denoting possible interaction with *SUT*. The test execution phase is realized by sending input values to follow a path leading to a *PASS* verdict and by observing outputs sent by *SUT*. Successions of computed inputs and observed outputs form traces. Those traces are evaluated with respect to their belonging to sets of traces of each path of the test case. Verdicts are assigned as soon as a trace belongs to the set of traces of a path whose last state is labeled by a verdict: the verdict emitted is of course the one associated with that last state. Without getting into details, let us underline that generated test cases are necessarily deterministic, which explains that there is no equivalent to our notion of *WeakPASS*. Such approaches are very useful to exploit as much as possible the knowledge of experts in the testing process, and thus to identify all the subtle properties to be tested. However, they need to be complemented by more automatic techniques: this is often a necessity because manually characterizing all the “interesting properties” to be tested becomes hard, if not impossible, as soon as the model is big. We have proposed several such techniques.

Coverage criteria based test purposes. The first technique is based on the idea of using coverage criteria,

as has been done on programs. In our context, a coverage criterion can be understood as a stopping criterion in the symbolic execution process, yielding a finite subtree of the whole symbolic execution. A trivial example of such a coverage criterion is the *transition coverage*. A subtree satisfies that criterion if and only if, for all transition of the *IOSTS* to be covered, there exists a symbolic path including that transition in its definition. We now discuss a more elaborated criterion, called the *inclusion criterion* [RAP 03], that we have proposed to use for selecting test purposes [GAS 06]. In order to give insight into how this criterion allows us to cut in the whole symbolic execution tree. Note that any arbitrary long behavior of an *IOSTS* can be understood as a sequence of “basic” behaviors. For example, in the ATM system of Example 1 basic behaviors are: (1) providing the user with money, (2) receiving deposit from the user or (3) giving the current level of the user account. Any “complex” behavior of the ATM system can be seen as a sequence of such basic behaviors. Now if we consider the symbolic execution of the ATM system, we would observe a lot (or even an infinite number) of occurrences of those basic behaviors. In other words, information on symbolic behaviors provided by the symbolic execution may be highly redundant in terms of basic behaviors. We propose cutting the symbolic execution of an *IOSTS* in order to lower this redundancy. Definition 12 of symbolic execution shows that behaviors are indeed determined by states, that is why our procedure to cut the tree is grounded on a relation upon states. From a symbolic state $\eta = (q, \pi, \sigma)$ we can extract constraints on the set A of attribute variables: the set of all possible interpretations $\nu_A : A \rightarrow M$ corresponding to η are restrictions⁶ to A of all interpretations $\nu : A \cup F \rightarrow M$ such that⁷ $M \models_\nu \bigwedge_{x \in A} (x = \sigma(x)) \wedge \pi$. If we consider two symbolic extended states η_1 and η_2

6. As usual, the restriction of an application $f : X \rightarrow Y$ to a subset Z of X will be denoted by $f|_Z$.

7. When reading $x = \sigma(x)$ for $x \in A$ in the formula, the reader should be aware that $\sigma(x)$ in fact denotes an expression in terms of variables of F .

built over the same state of an *IOSTS* and such that the set of possible interpretations of A for η_1 is included in the one of η_2 we says that $\eta_1 \subseteq \eta_2$. Figure 4.2 corresponds to a restriction by inclusion of the symbolic execution of the ATM system. Indeed, $\eta_4 \subseteq \text{init}$ since η_4 contains the same state q_0 as *init* and the constraints in η_4 , i.e. $\pi_0 = cp_0 \leq 2 \wedge x_1 \leq m_0 \wedge m_0 \geq 1000$, are stronger than those in *init* (*true*). The symbolic extended states η_3, η_5, η_6 and η_7 are handled in the same way.

Structuring based test purposes. Using off-the-shelf components to build new systems becomes more and more usual. For such systems, the testing approach often consists of testing basic components regardless of potential systems that will be built from them (this is called *unitary testing*), and then, for each system built on them, in focusing on testing properties concerning how those components collaborate (this is called *integration testing*). The main drawback of such an approach is that behaviors of a component tested at the unitary level may have few links with the ones really occurring in a the frame of a particular system using that component. Therefore, while integration testing should be dedicated to expose faults concerning collaboration between components, it may easily expose faults which are in fact due to defect of used components. Consider for example a component implementing usual operations $+$, $-$, $/$, and $*$. Consider now a system reusing components to compute average marks of students. Marks range between 0 and 20. While each operation may have been tested during the unitary phase, there are very few chances that the defined test cases cover the crucial property (from the system point of view): when adding n numbers between 0 and 20 and dividing the sum by n , that computes the average of the sum. It may happen that the component specifically fails for such combination of $+$, $/$, and numbers between 0 and 20. To summarize, in the context of our system, unitary testing of $*$ and $-$ is useless, while $+$ and $/$ may have been insufficiently tested, or at least not in the good context. In [FAI 07], we have addressed that problem

by proposing test purposes definition mechanisms allowing to define test purposes for basic components, from the knowledge of systems reusing them. The goal is then to reinforce unitary testing all along the life cycle of reused components. System models are represented as structured basic *IOSTSs*, each of them denoting a model of some basic component of the system. System models are symbolically executed as discussed in this chapter, using coverage criteria such as the inclusion criterion. A projection operator allows us to associate with any symbolic path of the system model, corresponding symbolic paths of each of basic components. Such a symbolic path associated with a basic component, may be seen as an over-constrained path of the symbolic execution of the component. The over-constraints come from the system model. Such over-constrained paths can then be used as unitary test purposes.

Refinement based test purpose. Depending on the level of abstraction of the model denoting the system, applying coverage criteria-based techniques for generating test purposes will lead us to define different test purposes. Intuitively, the more abstract the model, the generated test purposes are closer to properties defined at the requirement level. Conversely, the more concrete the model, the more the test purposes take into account implementation choices and are thus lucky to ensure a good level of coverage of the actual realization. When a design process is based on some refinement processes by introducing several models successively making explicit implementation choices, it is thus useful to use all of these models as starting points to built test purposes. In [FAI 08], we have proposed such a methodology in the particular case of *action refinement* [GOR 01]. That methodology is an extension of works presented in [BIJ 05] to symbolic issues. Action refinement relates abstract communication actions to some concrete action sequences. The refinement process consists of building concrete *IOSTSs* from more abstract ones by replacing

abstract communication actions by those action sequences. We show how to concretize test purposes extracted from abstract models, by concretizing the communication actions they involve at the level of abstraction of the system under test. Such concretized test purposes are directly usable by the algorithm presented in this chapter.

4.5.2. Implementation issues

The work presented here is implemented as an extension of the *AGATHA* tool set [LUG 01, RAP 03] which uses symbolic execution techniques to debug and validate models. The *AGATHA* tool allows us to unfold *IOSTS* models in the form of trees provided with path conditions for all paths of trees. Trees are computed according to coverage criteria including those grounding test purpose definitions discussed in section 4.5. Those test purposes are thus obtained for free. All rules defined in section 4.4.3 are implemented. However applying those rules does not necessarily lead to a deterministic process. Implementing deterministic strategies for rules appliance is still an open issue. Presburger arithmetic [PRE 29] constitutes the data part of *IOSTS* treated by *AGATHA*. The algorithm requires some decision procedures (for inclusion criterion) and constraint solving (to compute stimulations). This is done thanks to the Omega Library [OME 94].

4.6. Bibliography

- [BEL 05] BELINFANTE A., FRANTZEN L., SCHALLHART C., “Tools for Test Case Generation”, BROU M., JONSSON B., KATOEN J., LEUCKER M., PRETSCHNER A., Eds., *Model-based Testing of Reactive Systems: Advanced Lectures*, vol. 3472 of *LNCS*, Springer Verlag, 2005.

- [BIJ 05] VAN DER BIJL H. M., RENSINK A., TRETMAIS G. J., “Action Refinement in Conformance Testing”, KHENDEK F., DSSOULI R., Eds., *Testing of Communicating Systems (TESTCOM)*, Lecture Notes in Computer Science, Berlin, 2005, Springer Verlag, p. 81–96.
- [CLA 76] CLARKE L.-A., “A system to generate test data and symbolically execute programs”, *IEEE Transactions on Software Engineering*, vol. 2(3), 1976, p. 215–222.
- [FAI 07] FAIVRE A., GASTON C., LE GALL P., “Symbolic Model Based Testing for Component Oriented Systems”, SPRINGER BERLIN / HEIDELBERG, Ed., *Testing of Software and Communicating Systems TestCom / FATES 2007*, vol. 4581/2007 of *Lecture Notes in Computer Science*, 2007, p. 90–106.
- [FAI 08] FAIVRE A., GASTON C., LE GALL P., TOUIL A., “Test Purpose Concretization through Symbolic Action Refinement”, SPRINGER BERLIN / HEIDELBERG, Ed., *Testing of Software and Communicating Systems TestCom / FATES 2008*, vol. 5047/2008 of *Lecture Notes in Computer Science*, 2008, p. 184–199.
- [GAS 06] GASTON C., GALL P. L., RAPIN N., TOUIL A., “Symbolic execution techniques for test purpose definition”, *Testing of Communicating Systems: 18th IFIP TC 6/WG 6.1 International Conference, TestCom 2006. Lecture Notes in Computer Science*, New York, NY, USA, May 16–18 2006, Springer.
- [GOR 01] GORRIERI R., RENSINK A., “*Handbook of Process Algebra*”, Chapter “Action refinement”, p. 1047–1147, Elsevier, 2001.
- [JAR 04] JARD C., JÉRON T., “TGV: theory, principles and algorithms, A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems”, *Software Tools for Technology Transfer (STTT)*, vol. 6, 2004, Springer.
- [JEA 05] JEANNET B., JÉRON T., RUSU V., ZINOVIEVA E., “Symbolic Test Selection based on Approximate Analysis”, *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 3440, Edinburgh, April 2005.
- [KIN 75] KING J.-C., “A new approach to program testing”, *Proceedings of the international conference on Reliable software, Los Angeles, California*, vol. 21–23, 1975, p. 228–233.

- [LUG 01] LUGATO D., RAPIN N., GALLOIS J.-P., “Verification and tests generation for SDL industrial specifications with the AGATHA toolset”, PETTERSON P., YOVINE S., Eds., *Proceedings of the Workshop on Real-Time Tools Affiliated to CONCUR01*, Department of Information Technology UPPSALA UNIVERSITY Box 337, SE-751 05 Sweden, August 2001, ISSN 1404-3203.
- [OME 94] OMEGA, The Omega Project: Algorithms and Frameworks for Analyzing and Transforming Scientific Programs, 1994.
- [PRE 29] PRESBURGER M., “Über die Vollständigkeit eines gewissen Systems der Arithmetik”, *Comptes rendus du premier Congres des Math. des Pays Slaves*, 1929, p. 92-101, 395.
- [RAM 76] RAMAMOORTHY C.-V., HO S.-F., CHEN W.-T., “On the automated generation of program test data”, *IEEE Transactions on Software Engineering*, vol. 2(4), 1976, p. 293-300.
- [RAP 03] RAPIN N., GASTON C., LAPITRE A., GALLOIS J.-P., “Behavioural unfolding of formal specifications based on communicating automata”, *Proceedings of First Workshop on Automated Technology for Verification and Analysis*, Taiwan, 2003.
- [TRE 96a] TRETMAINS J., “Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation”, *Computer Networks and ISDN Systems*, vol. 29, 1996, p. 49–79.
- [TRE 96b] TRETMAINS J., “Test Generation with Inputs, Outputs and Repetitive Quiescence”, *Software—Concepts and Tools*, vol. 17, num. 3, 1996, p. 103–120, Springer-Verlag.
- [TRE 08] TRETMAINS J., *Formal Methods and Testing*, vol. 4949/2008, “Model Based Testing with Labelled Transition Systems”, p. 1-38, Springer Berlin / Heidelberg, 2008.

Chapter 5

Using MARTE and SysML for Modeling Real-Time Embedded Systems

5.1. Introduction

The design of embedded systems is a complex process that depends more and more on the effective interplay of multiple disciplines, such as mechanical, control, electronics and software engineering. In particular, the lack of a common design language between different disciplines hampers reasoning about system properties. The architecture of a system is particularly vulnerable to bad design choices made in the early design phases, which, unfortunately, often tend to show up later during the integration or construction phases. Designers of one part of the system may make incorrect assumptions concerning some other parts resulting in increasing development costs due to long feedback cycles.

Chapter written by Huascar ESPINOZA, Daniela CANCILA, Sébastien GÉRARD and Bran SELIC.

The use of models throughout the design process is gaining momentum in addressing these issues [SEL 07]. Models allow designers from different disciplines to share knowledge, facilitate design comprehension, and assess system-level trade-offs seeking higher quality and reliability. We subscribe to the view that both system design and integration will be reduced significantly by the use of a common modeling formalism, even for smaller projects. In particular, we believe that the widespread acceptance of UML (Unified Modeling Language) [OMGa] by industry and the use of UML profiles for domain-specific expressiveness ease the challenge considerably. A UML profile is the standard mechanism to create domain-specific modeling languages by still reusing a common set of well-known concepts and, more significantly, the corresponding tools.

A number of UML profiles have been proposed for modeling embedded systems, both within a standardization context and as research outcomes [JOH 07]. In our work, standardization is a crucial concern since it promotes lower overall training costs and helps to reduce the risk of being dependent on a single tool vendor. We particularly focus on two standard UML profiles that cover, as a whole, a broad cross-section of the modeling capabilities required for the embedded system domain. On the one hand, SysML (Systems Modeling Language) [OMGb] provides constructs to specify traceable requirements, structure and behavior of system blocks, as well as a parametric formalism to specify equation-based analytical models. On the other hand, MARTE (Modeling and Analysis Real-Time and Embedded systems) [OMGc] deals with time- and resource-constrained aspects, and includes a detailed taxonomy of hardware and software patterns along with their non-functional attributes to enable state-of-the-art quantitative analyses (e.g. performance and power consumption).

A major impediment to any kind of real-world application is that a single profile may not be sufficient to capture all aspects in the multidisciplinary domain of embedded systems. A number of industrial and research efforts have started to consider the use of both profiles in a synergistic manner to cover as much as possible the description of embedded systems at different abstraction levels (e.g. [SAT], [LAM 08], [INT], [ALB 08]). However, even in the standards world, different profiles may be mutually inconsistent and may overlap in ways that are not fully documented. Hence, it is essential to investigate ways of combining these two UML profiles to avoid syntactical and semantic conflicts and mismatches.

In this chapter, we provide the basis for a comparison between the two profiles. The purpose is to identify some typical scenarios in which their combined usage is of relevant added value in the embedded systems domain and, to provide a convenient starting point for those interested in using both profiles in a complementary manner. One problem is that, because they are constructed for different purposes and follow different design rationales, they tend to define different syntaxes for the same modeling concepts. This issue immediately puts profile users in a dilemma when they try to exploit both profiles in the same system model. Some minimum alignment is necessary to deal with such overlaps. Consequently, another objective of this Chapter is precisely to encourage the SysML and MARTE standardization task forces to provide a convergence and alignment program for their respective technologies.

The remainder of the paper is organized as follows. Section 5.2 outlines SysML and MARTE and their respective modeling capabilities. Section 5.3 introduces some anticipated scenarios that combine concepts from both expressiveness domains. In section 5.4, we provide some strategies to properly compare and integrate common

modeling constructs by taking into account the UML profiling capabilities. Section 5.5 discusses the contributions and shortcomings of other attempts at combining both profiles. A short discussion and conclusions round out the chapter.

5.2. Background

5.2.1. *UML profiling capabilities*

Due to the diverse nature of the disciplines involved in embedded system design, it is clear that a single modeling language is not suitable for describing all the different aspects. In this context, UML is often contrasted with domain-specific modeling language (DSML) approaches that create a new language from scratch [GRA 07]. The latter approach has the obvious advantage of enabling the definition of a language that is optimally suited to the problem at hand.

At first glance, this may seem the ideal approach to modeling language definition, but closer examination reveals that there can be serious drawbacks in this approach. If indeed each discipline is going to have its specific language, the problem will be how to interface the various parts of the design so that the integrated system can be verified, tested, or simply unambiguously understood. Furthermore, there is the problem of providing industrial-strength tools and training for a new custom language, which can result in significant and recurring expenses.

On the other hand, although UML was designed to eliminate the accidental complexity stemming from gratuitous diversity [SEL 04], it still provides a built-in mechanism, *profiles*, for creating DSMLs that can take advantage of existing UML tools. This is not to say that UML profiles avoid DSML integration problems, but many of

the *fragmentation problems*¹ [SHO 03] stemming from diversity can be mitigated. There is typically a lot of commonality even between different disciplines in embedded system design. For instance, the concepts of package, composition, property and connector, which are provided by UML, are common in many disciplines. The profiling mechanism is restricted to the use and extension of the existing UML language by adding more refined concepts and semantics that are consistent with the base UML semantics. A *stereotype* is the basic feature in profiles. It can be viewed as the specialization of an existing UML concept, which provides capability for modeling more refined domain-specific concepts or patterns. Stereotypes may have attributes and be associated with other stereotypes or existing UML concepts. From a notational viewpoint, stereotypes can give a different graphical symbol for UML model elements. For instance, a *class* model element stereotyped as “clock” might use a picture of a clock symbol instead of the ordinary class symbol.

The numerous claims made in literature about UML applicability as a DSML, contrast sharply with the small amount of published material on how UML profiles should be transparently used as DSMLs. We can distinguish two main categories of UML profiles [SEL 04]:

a) Specification profiles, which are fully-fledged DSMLs used to model systems from the viewpoint of a particular domain. SysML is an example of this kind of profile.

1. This is used to refer to the situation that occurs when different domain-specific languages are used to describe different aspects of a complex system. For example, one language might be used to describe the user interface function while a different one might be used for the database management and access functions. The individual languages involved could have very different models of computation, which raises the question of how to meld the different specifications into a coherent and consistent whole.

b) Annotation profiles which are used to add supplementary information to various kinds of UML elements that can then be interpreted by specialized tools or domain experts for various purposes, such as model analysis or code generators. Some parts of the MARTE profile, namely the sub-profiles that support analysis modeling, are examples of this latter category.

While the first category of profiles is generally well understood, some discussion is necessary to understand the second category. Particularly in the MARTE's analysis-specific sub-profiles (see section 5.3.4 on engineering/quantitative analysis), there is not a one-to-one mapping between analysis viewpoint concepts and UML concepts. The same analysis concept may be manifested in a number of different ways in a particular model. From the analysis viewpoint, all of these manifestations represent the same thing. Consequently, it must be possible to apply the same analysis stereotype to different base UML concepts, and conversely, different stereotypes (possibly from different analysis viewpoints) may be applied in the same model element.

For instance, the MARTE parts that support quantitative analysis can be applied to make a model look like an analysis model (e.g. a performance model) by tagging appropriate elements of the original UML model to represent concepts from the analysis viewpoint. These can then be used by an automated performance analysis tool to determine the fundamental performance properties of a software design. At the same time (and independently of the performance modeler) a reliability engineer might overlay a reliability-specific view on the same model to determine its overall reliability characteristics, and so on. This feature allows the same model to be viewed from different viewpoints (e.g. schedulability, performance, security, availability or timing). The ability to dynamically apply and un-apply a UML profile

without affecting the underlying model is crucial to this type of profile usage.

5.2.2. SysML and MARTE modeling capabilities

SysML and MARTE consider characteristics of the embedded systems domain at different abstraction levels, architectural styles, and particularly for specific purposes or application areas. In this section, we summarize their major modeling capabilities.

SysML is a UML profile “for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities” [OMGb]. The so-called *Block* concept is the common conceptual entity that factorizes many different kinds of system elements such as electronic or software components, mechanical parts, information units, and whatever structural entity composing the system under interest. Blocks articulate a set of modeling perspectives enabling separation of concerns during system design. Eschewing excessive detail², we identify the following key contributions of SysML regarding UML:

- *Architecture organization*: these include modeling concepts to organize system architecture descriptions as defined by the IEEE 1471 standard [EME 08]. Among them, the concepts of *view*, *viewpoint*, and *rationale* are the most important.

- *Blocks and flows*: block description and internal block diagrams of SysML enable the specification of more generic interactions and phenomena than those existing just in software systems. This includes physical flows such as liquids, energy or electrical flows. The dimension and

2. Further information on SysML publications, tutorials, tools, and links can be found via <http://www.omg-sysml.org/>.

measurement units of the flowing physical quantities can be explicitly defined.

- *Behavior*: although most behavior constructs in SysML are similar to UML (interactions, state machines, activities, and use cases), SysML refines some of them for modeling continuous systems and probabilities in activity diagrams.

- *Requirements*: SysML provides an explicit facility for modeling system requirements, along with their traceability with regard to the architecture evolution. These can be specified in either graphical or tabular format.

- *Parametrics*: a perspective called parametric diagram allows SysML users to describe, in a graphical manner, analytical relationships and constraints, such as those described by mathematical equations. Parametric diagrams provide a mechanism for integrating SysML design models with engineering analysis such as performance and reliability analyses.

MARTE is a UML profile that supports specification of real-time and embedded systems [OMGc]. In addition to functional design, this profile adds constructs to describe the hardware and software (e.g. OS services) resources and defines specific properties to enable designers to perform timing and power consumption analysis. With regard to UML, MARTE adds the following features³:

- *NFPs*. The NFPs (non-functional properties) modeling framework provides means to declare, qualify, and apply semantically well-formed non-functional properties (e.g. throughputs, bandwidths, delays, memory usage), supported by a language to formulate algebraic and time expressions.

- *Time*. A highly refined model of time and timing mechanisms integrates concepts from different sub-domains

3. Further information on MARTE publications, tutorials, tools, and links can be found via <http://www.omgmarTE.org/>

in embedded systems design, such as causal time, synchronous time, and chronometric time.

- *Software application.* A common model of computation provides semantic support for the real-time object paradigm. This paradigm allows applications to be specified at a high abstraction level, by delegating concurrency, communication, and time-constraint aspects to a modular unit called *real-time unit* (RtUnit).

- *Components.* The MARTE component model extends UML composite structures and SysML internal block diagrams with a notion of message-based communications. This is intended to support the request-reply/publish-consume communication paradigm.

- *HW/SW resources.* Software and hardware resources can be described at different levels of abstraction, including their typical services, as found in common OS platforms, and common non-functional properties like power consumption or memory usage.

- *Quantitative analysis.* A set of pre-defined non-functional annotations enable MARTE models to bridge with state-of-the-art performance and scheduling analysis tools.

5.3. Scenarios of combined usage

To focus our study, we identify a set of representative scenarios in which a combined usage of SysML and MARTE is of relevant added value in the embedded systems domain. Although this set is certainly incomplete, it allows us to drive our comparison framework in a more focused manner. The intent is to adequately answer the question of what can each profile target best in modeling, and then determine their possible integration issues.

5.3.1. *Defining architecture frameworks*

The modeling capabilities of both SysML and MARTE are rich enough for a wide range of design approaches. This has the flexibility for supporting and integrating multiple design perspectives, but also the difficulty of understanding and choosing among a variety of language alternatives. In both cases, there is not a predetermined way to use the language constructs through the development lifecycle. This means that a consistent modeling framework and methodology should be defined for using these profiles in a particular application domain.

Architecture organization

In the IEEE 1471 standard (and in the draft of its upcoming update ISO/IEC 42010), the concept of modeling framework is referred as *architecture framework*. An architecture framework “establishes a common practice for creating, organizing, interpreting and analyzing architectural descriptions used within a particular domain of application or stakeholder community” [EME 08]. An architecture framework identifies one or more predefined architectural viewpoints. Viewpoints define how to construct views, which are in turn a representation of a system from the perspective of a set of modeling concerns.

SysML implements IEEE 1471 by providing a set of constructs to organize models. In particular, SysML does not define any specific viewpoint, but it provides means to specify how views are built, and to relate any user-specific view to a given viewpoint. This is aligned with the IEEE 1471 approach that envisages libraries of viewpoints, in order to enable architects selecting those useful for system design at hand.

Although MARTE does not provide any concrete model element to define viewpoints, it has an implicit conception of

viewpoints rooted in its design rationale. Indeed, some of the MARTE constructs have been designed to define domain-specific viewpoints (see section 5.2.1). Such viewpoints, when applied to a standard UML model, cast that model in a domain-specific way and may also add supplementary information to the model relevant to the viewpoint.

In consequence, there is no language overlapping in this respect. SysML and MARTE can be used in complementary way. While SysML provides means to create viewpoints in a general way, MARTE provides particular viewpoints. However, an open issue is to enable designers of architecture frameworks to build consistent inter-view rules that ensure meaningful and correct-by-construction models.

5.3.2. Requirements engineering

System usage scenarios

Requirements engineering is the process by which the requirements for systems and software products are gathered, analyzed, documented and managed throughout the development life cycle. UML has traditionally been used to document user requirements by means of use case diagrams. Use cases follow a graphical, scenario-based approach. This means that requirements are organized into system usage histories, acting as a user-friendly bridge between technical and business stakeholders.

Although use cases may be formalized to a certain degree, for example by using sequence diagrams in order to detail such usage histories, they are often criticized for a number of limitations. For instance, use cases lack well-defined semantics, which may lead to differences in interpretations by stakeholders [SOA 08]. They are applied mainly to model functional requirements, but are not very helpful to model non-functional ones. Also, relationships between requirements and the various architectural parts that satisfy

those requirements are difficult to trace. SysML and MARTE provide some significant enhancements in these aspects.

Requirements management / traceability

SysML requirements diagrams explicitly show the various kinds of relationships between different requirements. This enlarges the spectrum of requirements engineering tools that can interact with UML tools. In effect, the SysML requirements modeling constructs are intended to provide an automated bridge between architectural models and traditional requirements management tools such as, for instance, Requisite Pro, Rectify or DOORS [ALB 08]. The latter provide support for traceability analysis, flow-down, derivation, assignment, among other requirement engineering activities. In particular, requirements tracing is very useful, for example, to identify how requirements are affected by changes, what is the purpose of a requirement, and to prioritize requirements. Traceability also provides a possibility of verifying whether or not all requirements have been fulfilled by the system and sub-system components.

Non-functional requirements

On its side, MARTE offers key features to specify non-functional requirements in general and timing requirements in particular. In embedded systems development, non-functional characteristics (e.g. performance, reliability, power consumption) influence a wide range of design decisions [CAN 08]. One possible scenario is using MARTE annotations to characterize non-functional constraints in use case diagrams and their underlying sequence diagrams. This provides two important capabilities leading toward more formal requirements specification.

First, non-functional requirements are cohesively specified along with functional requirements. While specifying non-functional aspects is possible with SysML requirements diagrams, their semantic relationship to

concrete functional system usages is hard to capture. In particular, the completeness of requirements satisfaction in real-time systems is strongly dependent on the coupling between system function and timing. In MARTE, timing annotations provide semantic definitions closely related to the system behavior. For instance, we can define a jitter constraint in the arrival of an event and identify if such an event relates either to a send, receive or consume occurrence within a sequence diagram.

Second, non-functional annotations follow a well-defined textual syntax, which is supported by the MARTE's Value Specification Language (VSL). The main advantages of this level of formalization are the ability to support automated validation, verification, traceability, and, more simply, an unambiguous understanding by stakeholders.

Clearly, SysML and MARTE concepts, articulated by use cases and scenarios, are highly complementary. While scenarios are useful for managing change and evolution, managing scenario traceability across multiple changes becomes increasingly difficult. SysML contributes with constructs to define such traceability relations. Additionally, MARTE completes scenario precision with well-formed non-functional annotations. However, it is important to define clear consistency rules to combine them in a typical development process using different requirements engineering tools.

5.3.3. System-level design integration

In a typical development process for embedded systems, software and other forms of engineering will be at least partially concurrent. The system is developed by composing pieces that, all or in part, have already been pre-designed or designed independently by different teams specialized in different disciplines. This is often done in vertical design

chains such as, for example, in the avionics and automotive industries. Therefore, there is a need to support design artifacts by common and standard specification formalisms that will allow plug-and-play of subsystems and their implementation [SIF 05]. This has a particular impact for improving the quality of the system architecture.

A model view is a typical abstraction that helps to divide a complex problem into smaller and comprehensible parts. In order to integrate global models, e.g. for performing system-level analysis, we must recombine these smaller parts in a consistent way. UML supports model composition by means of *composite structure diagrams*. The basic principle is to define usages of model elements in a given context. The idea of composite system models is to describe how information from multiple modeling artifacts and views is to be joined, deployed or configured. Although there is a linguistic divergence⁴, both SysML and MARTE reuse this notion with some particularities. Thus, some aspects need to be taken into consideration for their combined use.

Hierarchy and composition

To understand the pragmatic problems of SysML-MARTE joint usage, let us consider the scenario of a large development project with engineers from multiple disciplines. It should be carefully decided how the system model will be created by integrating the models from different disciplines. One important issue is the layering and mismatched sub-system hierarchies, which has been comprehensively addressed by Maier [MAI 06. For instance, in multiprocessor software-intensive design, the electronic system perspective typically represents a hierarchy of interconnected processors, each containing software units.

4. While SysML uses the “term block” for such composition units, MARTE called it “structured component”.

From the software perspective, the hierarchy is reversed, as generally illustrated in MARTE examples [OMGc]. At the top is a distributed application, composed of software units that interact through data- and message-based interfaces. Below the application are the operating system (OS) and library layers that support the distributed application. At the bottom of the hierarchy, the hardware (processors and interconnection networks) completes the model.

This aspect is important when deciding which kind of modeling constructs will be used to represent hierarchy, allocation/deployment and composition. For instance, while a composition relationship would be used for the hardware viewpoint, an allocation relationship (supported by both SysML and MARTE) would be preferred by the software designers. Some compatibility or merging rules need to be defined to provide system-level consistency.

While in some cases, engineers from a given discipline would exclusively use either SysML or MARTE, in other cases they would need to combine concepts from both profiles. An integration scenario may consist of starting from a system-level model, probably specified with SysML blocks, and adding later some additional semantics to some of these blocks, for example by applying MARTE stereotypes. Indeed, the detail level underlying MARTE constructs makes it possible to specify some aspects such as concurrency and synchronization mechanisms, as well as resource patterns such as processing resources, communication buses, or power supply devices along with a set of predefined quality attributes. This is especially required in application areas where designers are interested in preparing models to perform simulation, quantitative analysis or product synthesis.

Interfacing/interaction

A central concern in system and software architecting is to understand the interfaces and interactions between structural elements. The nature of such interfaces and interactions can significantly vary from software to other kind of systems. Looking at the structural aspects, we can see that MARTE adopted the notions of *port* and *flow* from SysML. This may seem very convenient from a perspective of semantic consistency. However, SysML flow ports require careful attention when used to model flows of physical quantities, such as for example energy or torque. Care must be exercised in defining explicit behavior on flow transmission. SysML physical flows are often continuous in time, whereas MARTE flows are used to describe data transmission with particular delegation semantics. While providing a precise semantics to flows is currently outside the scope of both profiles [CUC 08], their combined use should define a common “semantic envelope” that could be shared by SysML and MARTE. In this way, composing models from different disciplines (and probably using stereotypes from different profiles) will preserve system-level consistency.

5.3.4. Engineering/quantitative analysis

Engineering analysis (SysML term) or quantitative analysis (MARTE term) concern the use of mathematical techniques to study certain quality attributes of the system. They include stress, thermal or fluid analysis in mechanical engineering, and performance or reliability analysis in software engineering. One challenging problem in model-based engineering is to integrate models that are commonly used for system production or software code generation with the information that is relevant to perform analysis [ESP 08]. The goal is to reduce the time required to prepare a design model for performing analysis and to ensure greater

accuracy of an analysis model by directly associating it with the actual system model. Both SysML and MARTE provide key contributions in this direction, but some alignment work has still to be done.

Timing modeling

Beyond the annotation of quality attributes, timing analysis requires a careful semantic definition closely related to the system behavior and the different models of computation and communication [AND 07].

SysML does not extend the UML time model, but a set of preliminary requirements were established by its standardization board, including continuous time models and relativistic effects that can occur in distributed systems.

In MARTE, time modeling is a core concern. We can distinguish at least three layers of time constructs organized by their level of complexity:

- In the first layer, time is presented as a set of fundamental notions such as time instant, duration, time bases, or clocks. These provide an unambiguous basis to express further modeling constructs and well-formed value spaces for data types.
- In the second layer, MARTE provides mechanisms to annotate timing requirements and constraints in UML models. One key modeling feature is the concept of observation, which is specialized in instant and duration *observations*. Observations provide marking points in UML models to specify assertions. Some typical assertions have been embedded in ready-to-use patterns, such as, for example, jitters.
- In the third layer, time concepts are defined as part of the behavior, not mere annotations. This set of constructs covers both physical and logical time. While the logical time is the basis to understand basic temporal notions, this is

further refined to support the precedence/dependency in presence of concurrency, and clocked time abstractions to cover synchronous language abstractions (such as those from Lustre, Signal or Esterel).

While the adoption of the two basic layers is certainly useful for system engineering in general, the third layer would need some extensions to include, for example, modeling of the continuous dynamics of systems [JOH 07]. This would need to provide means to specify system behavior in terms of hybrid discrete event and differential algebraic equation systems.

Quantities values

In SysML, a value property represents a quantifiable characteristic of a block (e.g. energy consumption, surface, and temperature range of a microprocessor). Value properties are defined in block compartments by assigning a name and a value type. A value type is a kind of data type that carries a particular pair consisting of a dimension and a measurement unit.

For its part, MARTE uses its non-functional properties (NFPs) modeling framework [ESP 06]. The NFP modeling framework provides the ability to encapsulate rich annotations within non-functional values. For instance, consider a property named “latency”. Instead of specifying its meaning in an axiomatic way such as: “duration in milliseconds with an accuracy of 0.01 measured by simulation as a mean value”, the specification itself includes all this information in a normalized syntax. For this purpose, the MARTE data type system includes the required data structure (value, unit, precision, measurement source, etc.) in a predefined library. For example, Duration, DataSize, DataTxRate, Frequency and Power are typical non-functional data types. Different units of the same physical quantity may be transformed into, or expressed in terms of,

existing base units through a given conversion factor and an offset factor.

One of the main issues when trying to combine both profiles is that the modeling approaches to declare and specify quantitative values is quite different. The main difference is that SysML hard coded the qualification of value types with the stereotypes unit and dimension, while MARTE allows for declaring a set of qualifiers as an extendable library. As a consequence, using both modeling mechanisms in the same model may lead to inconsistencies and cumbersome model processing. Alignment of these two modeling styles is a key issue that should be posted as a joint effort between the MARTE and SysML task forces at OMG.

Beyond syntactical issues, the debate should be centered on providing practical capabilities to both profiles. We believe that at least two key capabilities should be allowed from SysML and MARTE models:

- *Measurement conversion.* Quantities need to be expressed in different measurement units while still allowing tools to convert quantities from one set of units to another.
- *Dimensional analysis.* Physical expressions must guarantee the consistency of equations and solve resulting measurement units and dimensions.

If we look at SysML, it forces tools to be hard-coded with the transformations between measurement units (e.g., from “mm” to “m”) because unit definition lacks conversion factors. Furthermore, dimensional analysis is not possible in SysML since dimensions are not defined in terms of basic dimensions and their exponents (e.g. $F = LMT^{-2}$). Conversely, while MARTE supports unit conversion, the notion of dimension has not been considered at all.

Parameters/expressions

Parameterized expressions are a primary feature in order to prepare models for analyzing performance, risk, costs, and so on [ESP 08]. SysML parametric diagrams capture constraints among performance, physical and other quality-related properties of the system and its environment. Such constraints are specified as equations among value properties. Equations can be specified in a third-party language (e.g. MathML or Modelica).

The basic composite modeling entity is the Constraint Block. The relationships between modeling entities within a constraint block are not committed to an “input” or “output” role early. Thus, they are called non-causal, as opposed to data flow and control flow approaches. Non-causal models are suitable to enable analytic processing, and can increase the level of integration/automation between design tools and analysis tools.

In addition, MARTE’s VSL gives the syntax to formulate algebraic and time expressions. VSL is rooted in OCL. However, VSL was intended to provide more compact expressions. In addition, VSL extends arithmetic and logical expressions with time-related annotations, which can be extended by libraries providing new functions.

We believe that a combined use of SysML parametric diagrams and VSL would provide significant advantages. While parametric diagrams provide a user-friendly formalism to specify non-causal models, VSL provides the textual syntax for constraint expressions. One open issue in VSL is its extension to support special expressions used in system engineering. For instance, differential and integrals, continuous time expressions, and discrete event equations.

5.4. Combination Strategies

In this section, we outline some issues in combining SysML and MARTE and propose general strategies to integrate both profiles in a single modeling framework.

5.4.1. Issues

Table 5.1 summarizes the modeling aspects discussed in section 5.3 along with a set of profile combination cases and implementation issues, which are elaborated below.

Modeling concern (from section 5.3)	SysML concepts (examples)	MARTE concepts (examples)	(Conflicting) combination cases*	Implementation issues*
Architecture organization	view, viewpoint, rationale, etc.	-	-	library of MARTE viewpoints
Hierarchy/composition	block, part, allocation	component, parts, hw/sw patterns, allocations	(a)	(1) & (2)
Interfacing/interaction	port, flow, items	idem SysML + message-based	(a) (c)	(2)
Spectrum of behavioral Models	rate, continuous, discrete edges, probability, etc.	synchronous/asynchronous causal/real-time	(c), (d)	(1)
System usage scenarios	use case, sequence diagrams	use cases, sequence diagram	common UML concepts	-
Requirements processing/trace	requirement, trace relationships, test case	-	-	(1) SysML requirements can be fully imported

Non-functional requirements	requirement	nfp constraint, VSL expressions	complementary	(1)
Time modeling	(UML) time constraints	extended time constraints, clocks, predefined nfp's for time analysis	-	(1) & (2) not all MARTE time notions may be required
Quantity values	value property, value type, unit, dimension	nfp, nfp type, unit	(c) (d) overlapping	(2) language alignment required
Parameters/expressions	constraint blocks, parametric diagrams	VSL expressions	(c) complementary	(1) VSL as expression language

* see text for full explanations

Table 5.1. *MARTE/SysML combination issues*

Combination case

We can generalize typical categories of the combined usage of UML profiles as follows:

a) Each language is used for different partitions of the system, in which case they are practically mutually exclusive and conflicts are small or even negligible. For example, SysML is used for mechanical design and MARTE for software design. As shown in Table 5.1, this category needs special attention when defining the hierarchy/composition and interfacing/interaction constructs during a system-level integration phase.

b) Each language is used for a different level of abstraction. Again, there is not much conflict here. For instance, SysML is used for system domain analysis and MARTE for a detailed design.

c) The languages are used in combination into the same parts of a model (e.g. in the same modeling view) and for the same purpose or concern. For instance, we may use the SysML facilities for continuous behavior in activity diagrams and the MARTE time annotations to support performance analysis.

d) The languages are used in combination but for different purposes such as, for example, using MARTE annotations to do performance analysis on a SysML model. The UML profiling capabilities of being able to apply many stereotypes to a single model entity is crucial for this kind of usage. There may be some conflict in trying to keep the consistency between MARTE non-functional annotations embedded in stereotype attributes (e.g. performance analysis stereotypes), with other SysML specifications such as block quantity value annotations or block constraint parameters.

Implementation issues

The above combined cases may result in different combination issues from a tool implementation viewpoint. A supporting toolset that accompanies UML profiles is, strictly speaking, not a part of the language problem. However, the utility of a profile combination is directly related to the maturity of the supporting tools. We identify the following scenarios in combining MARTE and SysML profiles in modeling tools:

- 1) The simplest solution is to apply the profiles (i.e. the full profile definition) or sub-profiles (i.e. sub-packages stereotyped as profiles) where needed within a model. For example, a SysML user could specify that it requires the full Time Modeling package of MARTE. UML tools can manage this case because of the modularity defined in MARTE (organized in “extension units”) and the UML’s ability to select only those profile packages that are of direct interest.

2) While it is likely we may use some concepts of a profile or sub-profile, designers may not want to include the full profile or sub-profile package in their models. For instance, MARTE profile users may want to gain access to SysML concepts of the block, but they may prefer to use the MARTE constructs for flows. UML does not allow for applying single stereotypes (contained in a profile) into a model. What is needed is a decoupling/merging mechanism to compose profile concepts and to make it available for profile users. Managing semantic compatibility is a requirement here.

In general, a hypothetical MARTE-SysML modeling tool should allow for filtering appropriate information according to specific users. Some engineering disciplines may be satisfied with a high-level description (e.g. blocks-and-flows description), software developers may want detailed behavior specifications, while analysis experts may require information on a set of non-functional properties. This aspect is more relevant when more than one stereotype is applied to a single UML model element. For instance, we can consider a SysML Block, as a specific hardware resource by annotating it with the appropriate MARTE stereotype. However, it is considered as a “resource” from a software viewpoint, but not from an electronic viewpoint. This needs a suitable presentation mechanism to show the right stereotype for different stakeholders.

5.4.2. Strategies

Defining a modeling framework that combines SysML and MARTE requires a systematic comparison of the two. We consider that at least the following aspects should be assessed in such work:

1. *Conceptual Domain Coverage.* Beyond syntactical aspects, it is important to begin by assessing both profiles from a conceptual viewpoint. The intent is to reach an

overall understanding of these profiles and determine what application domains are best covered by each. A good starting point is using the conceptual domain models underlying the UML profiles. Conceptual domain models are created as free as possible from considerations related to specific solution technologies so as to not embody any premature decisions that may hamper later language use. Currently, the MARTE specification provides a conceptual domain model in the form of a metamodel with a textual description. On the other hand, SysML directly defined UML stereotypes extending the UML metamodel. Although a conceptual description is provided, a metamodel would significantly help in identifying/comparing conceptualization entities of the targeted domain.

2. *Semantic/Syntactic Overlapping.* The evaluation of related points between both profiles should be clearly identified by defining overlapping semantics (conceptual coverage), abstract syntax (extended UML constructs), and concrete syntax (symbols and terminology: synonymy/homonymy). The intent is to determine which aspects of both profiles can be consistently aligned and/or selected to consistently use both profiles. Overlapping aspects must be assessed in the light of one of the language use cases, (c) or (d), identified in section 5.3.1. While case (d) needs revisiting the notion of views and viewpoints in the context of UML profiles (see section 5.2.1), case (c) requires a more careful treatment of semantic consistency.

3. *Usability/Pragmatics.* Usability issues are concerned with concepts such as ease of use, productivity, and user satisfaction. Once the overlapping concepts are identified and before deciding which profile features to adopt in a given modeling framework, we should identify the effectiveness of different symbols or stereotype names for model understandability, as well as the number of steps needed to accomplish a modeling goal. Of course that may depend on a tools' maturity. However, syntactical design choices can help

avoid complicated ways of performing modeling steps or features which invite mistakes.

4. *Expressiveness Limitations.* One fundamental requirement that should drive a useful comparison is completeness and lack of model expressiveness. The evaluation of missing aspects needs to be objective by clearly identifying whether it implies a conceptual, semantic or attribute insufficiency. This raises the problems of improving and extending both profiles, which is an important goal of our research.

5. *Abstraction/refinement levels.* One fundamental difference between SysML and MARTE relates to their ontological considerations. For example, while SysML does not consider any “functional” classification of structural elements (only the generic concept of Block exists), MARTE goes deeper by providing a detailed taxonomy of application and resource structural elements (e.g. active/passive objects for the application, or processor/tasks resources for the platform). Using abstract or concrete language concepts will depend on the phase of development, and the kind of model processing (communication, simulation, verification, etc.) required at each level.

5.5. Related work

The academic and industrial communities have recently begun to investigate the complementary use of SysML and MARTE to support model-based development of embedded systems.

Among current projects in the embedded systems domain, MeMVaTE_x [ALB 08] defines a model-based methodology for modeling, validating and tracing system requirements. It is based on EAST-ADL, a modeling language dedicated to the automotive industry, but it also relies on SysML for requirements modeling and on MARTE for modeling timing

aspects. Since these aspects are practically independent, their combination is handled methodologically, by providing consistent rules on when and where to apply concepts of the individual profiles. Another project combining these two profiles is INTERESTED [INT], which attempts to create an interoperable tool-chain for enhanced rapid design, prototyping and code generation of embedded systems. This work aims at a more extensive use of SysML and MARTE. While the first profile serves to describe the high-level architecture organized around functional blocks, the second one provides the standard annotations to enable timing analysis. However, the methodological rules to guide the combined use of both profiles have yet to be established. Understanding where and how to apply SysML and MARTE concepts while ensuring semantic consistency is an open issue.

Two additional projects were recently started with the objective of adopting SysML and MARTE in the hardware/software co-design field. One of these, the SATURN project [SAT], proposes to bridge the gap between SysML/MARTE modeling and tools for architecture exploration, simulation and synthesis (in SystemC/VHDL for hardware and C/C++ for embedded software). The main strategy is to adopt most of the constructs of SysML and to integrate MARTE for adding the formal semantics of different models of computation and thus enable system verification. The second project, Lambda [LAM 08], intends to reconcile a number of related standards, including SysML, MARTE, AADL and IP-XACT, to develop a library of broadly used software and hardware platforms.

At the other end of the spectrum, there is very little research literature discussing integrated approaches for system and software modeling based on UML. One example is [HAU 08], where the authors evaluate how UML and SysML could be consistently used for both system and

software modeling. Perhaps the main contribution of this work is a mapping between SysML and UML concepts and the identification of the application domains associated with each concept. Unlike this work, we attempt to provide a more rigorous comparison of system and software modeling concerns, and additionally, enrich expressiveness with MARTE features.

With regard to the combination of profiles at tooling level, the authors in [BEN 08] introduce a packaging unit called MDATC (which stands for Model-Driven Architecture Tool Component) that serves to collect metamodels and/or profiles, know-how, and required resources in order to support domain-specific activities. Thus, by using MDATC, modeling rules and constraints in the use of multiple profiles can be represented and exchanged in a standard format. Currently, MDATC is under standardization at OMG.

One possible source of solutions to the general problem of composing mixed models can be found in related work done within the formal methods community. For example, in [FRA 07] the authors discuss heterogeneous and hierarchical composition of models to design embedded systems. The work is based on two principles. First, a model can be described at different levels of abstraction (hierarchy of models); second, each level of abstraction can be described by a different formalism (heterogeneousness of models). Therefore, a designer can choose their own formalism which fits in each level. Each model is formalized by a graph and the authors are able to prove a composition of models using basic notions of category theory.

We end this section by highlighting the relationship of composition of profiles to the so-called “fragmentation problem” [SHO 03]. For instance, an aspect-oriented approach supporting metamodel composition is proposed in [FRA 07]. The authors focus on implementing composition mechanisms for matching and merging model elements that

crosscut the dominant structure described in a primary model. The composition directives are implemented in Kermeta, an open-source metamodeling language. Even if language composition between different metamodels is certainly a more difficult problem than combining stereotypes extending the same metamodel, special care must be exercised. Our study can be inserted in this lively context and viewed as a modest contribution in the composition of profiles, with special focus on SysML and MARTE, although in general the fragmentation problem is left as an open problem.

5.6. Conclusion

Due to the varying nature of the disciplines involved in embedded system design, it is clear that a single modeling language, such as for example UML, may not be suitable for all aspects. We believe that the UML profile mechanism is well suited to create domain-specific languages, by providing a common semantic and syntactic foundation while also permitting reuse of the underlying modeling tools. Currently, there are an important number of profiles that may make their usage cumbersome, as they are often created mutually inconsistent and overlapping. In this Chapter we presented some integration strategies for combining the SysML and MARTE profiles. Both provide essential ingredients to model embedded systems. Our intent is to offer a better understanding of their conceptual domains, and to help in using both profiles in a single model by avoiding semantic and syntactical mismatches.

We presented some typical scenarios in which their combined usage is of relevant added value in the embedded systems domain. In general, using modeling constructs from one or the other profile depends on the expressive power a constructs should provide to practitioners. In a simple usage scenario, the intent may be to aid understanding and to

communicate about a system design. As such, it is not necessary to define a detailed description or precise semantics, and basic evaluations of the architecture could be performed. In a more elaborated scenario, however, we may be interested in using powerful analysis tools, simulators, model checkers, product synthesis tools, and the like. In this case, the necessary levels of specification detail and semantic precision are much higher. While both forms of specification have merit, their usage will be driven by the specific needs of a particular development process and its phases through the system life cycle.

Some of the future work that we envisage consists of providing a detailed comparison of SysML and MARTE's semantic and syntax, providing pertinent examples on their combined usage, and suggesting some improvements regarding language mismatches and limitations.

5.7. Acknowledgements

The authors of this chapter would like to thank Hubert Dubois for his valuable feedback on earlier drafts of this chapter. The work presented here was partially carried out within the System@tic competitiveness cluster projects Lambda and IMOFIS.

5.8. Bibliography

- [ALB 08] ALBINET A., BEGOC S., BOULANGER J.-L., CASSE O., DAL I., DUBOIS H., LAKHAL F., LOUAR D., PERALDI-FRATI M.-A., SOREL Y. and VAN Q.-D.. "The MeMVaTEx methodology: from requirements to models in automotive application design", *4th European Congress ERTS Embedded Real Time Software*. Toulouse, France, January 2008.
- [AND 07] ANDRÉ C., "Time Modeling in MARTE", in *FDL'07 Forum on Specification and Design Languages*, Barcelona, Spain, 2007.

- [CAN 08] CANCELA D., PASSERONE R., Functional and Structural Properties in the Model-Driven Engineering Approach, ETFA 2008.
- [BEN 08] BENDRAOU R., DESFRAY P., GERVAIS M.-P. and MULLER A., "MDA Tool Components: a proposal for packaging know-how in model driven development", *Software and System Modeling*, 2008, Vol 7, pp. 329-343.
- [FEN 08] FENG T.H. and LEE E., Scalable models using model transformation, technical report No. UCB/EECS-2008-85, 2008.
- [CUC 08] CUCCURU A., GÉRARD S., RADERMACHER A., "Meaningful composite structures - on the semantics of ports in UML2", *MoDELS'08*, September 2008.
- [EME 08] EMERY D. and HILLIARD R., "Updating IEEE 1471: architecture frameworks and other topics", *7th Working IEEE/IFIP Conference on Software Architecture WICSA*, 2008
- [ESP 06] ESPINOZA H., DUBOIS H., GÉRARD S., MEDINA J., PETRIU D., MURRAY C., "Annotating UML models with non-functional properties for quantitative analysis", *Lecture Notes in Computer Science*, Volume 3844, pp. 79-90. Springer-Verlag, January, 2006.
- [ESP 08] ESPINOZA H., SERVAT D., and GÉRARD S., "Leveraging analysis-aided design decision knowledge in UML-based development of embedded systems", *SHARK at ICSE'08*, Leipzig, May 2008.
- [FRA 07] FRANCE R., FLEUREY F., REDDY R., BAUDRY B., and GHOSH S., "Providing support for model composition in metamodels", *Proceedings of EDOC 2007*, Annapolis, MD, USA, October 2007.
- [GRA 07] GRAY J., TOLVANEN J.-P., KELLY S., GOKHALE A., NEEMA S., SPRINKLE J., "Domain-specific modeling" in *CRC Handbook of Dynamic System Modeling*, Paul A. Fishwick (ed.), CRC Press, 2007.
- [HAU 08] HAUSE M. and THOM F., "Building bridges between systems and software with SysML and UML", in *INCOSE Intl. Symposium*, 2008, June

- [INT] INTERESTED EU Project: Interoperable embedded systems tool-chain for enhanced rapid design, prototyping and code generation, <http://www.interested-ip.eu/index.html>
- [JOH 07] JOHNSON T., JOBE J., PAREDIS C., BURKHART R., “Modeling continuous system dynamics in SysML”, *Proceedings of the IMECE 2007*, November 2007.
- [LAG 08] LAGARDE F., ESPINOZA H., TERRIER F., ANDRÉ C., GÉRARD S., “Leveraging patterns on domain models to improve UML profile definition”, *FASE 2008*, pp. 116-130, Budapest, April 2008.
- [LAM 08] Lambda Project, Lambda Libraries for Applying Model Based Development Approaches, Technical Annex, May 2008
- [MAI 06] MAIER M., “System and software architecture reconciliation”, *Systems Engineering Journal*, 2006, pp. 146-159.
- [MUR 08] MURA M., PANDA A. and PREVOSINI M., “Executable Models and Verification from MARTE and SysML: a Comparative Study of Code Generation Capabilities”, *Proceedings of MARTE Workshop, Event Satellite of DATE'08 Conference*, March 2008, Munich, Germany.
- [OMGa] OMG, Unified Modeling Language, UML™ Superstructure, V2.1.2
- [OMGb] OMG, Systems Modeling Language SysML™, V1.0
- [OMGc] OMG, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2
- [SAT] SATURN Project: SysML bAsed modeling, architecTure exploRation, simulation and syNthesis for complex embedded systems, <http://www.saturnsysml.eu>
- [SEL 04] SELIC B., “On the semantic foundations of standard UML 2.0”, in *SFM-RT*, LNCS, pp. 181-199, 2004
- [SEL 07a] SELIC B., “From model-driven development to model-driven engineering”, *Keynote Talk at ECRTS'07*, July, 2007.
- [SEL 07b] SELIC B., “A systematic approach to domain-specific language design using UML”, *ISORC 2007*.

- [SHO 03] SHONLE M., LIEBERHERR K., SHAH A., “XAspects: An Extensible System for Domain-Specific Aspect Languages”, *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA’03*, Pages 28 - 37, October 26-30, 2003, Anaheim, California, USA.
- [SIF 05] SIFAKIS J., “Embedded systems – challenges and work directions”, *Principles of Distributed Systems*, LNCS 3544, 2005
- [SOA 08] SOARES M.S., VRANCKEN J.L.M., “A Proposed Extension to the SysML Requirements diagram”, *IASTED International Conference on Software Engineering*, Austria, 2008.

Chapter 6

Software Model-Based Performance Analysis

6.1. Introduction

Model-Driven Development (MDD) is an evolutionary step in the software field that is changing the focus of software development from code to models. MDD is based on *abstraction* to separate the model of the application under construction from underlying platform models, and *automation* to generate code from models. The emphasis on models facilitates the analysis of non-functional properties (NFP), such as performance, scalability, reliability, security, safety, etc. of the software under development based on its model. This brings more “engineering” into software development, leading to the paradigm known as Model-Driven Engineering (MDE).

Over the years, many formalisms and tools for the analysis of different NFPs have been developed, for example

Chapter written by Dorina C. PETRIU.

queuing networks, stochastic Petri nets, stochastic process algebras, fault trees, probabilistic time automata, formal logic, etc. The research challenge is to bridge the gap between MDE and existing NFP analysis formalisms and tools rather than to “reinvent the wheel”. An approach for the analysis of different NFPs composed of the following steps is emerging in the literature: a) add annotations describing the respective NFP to the software model, b) define a model transformation from the annotated software models to the formalism used for NFP analysis, c) analyze the NFP model using existing solvers, and d) give feedback to designers. Figure 6.1 illustrates this process for the case where performance is the analyzed NFP; similar “analysis loops” exist for other NFPs.

In the case of UML-based software development, the extensions required for NFP-specific annotations are defined as UML profiles, which provide an additional advantage to be processed by standard UML tools without any change in the tool support. Two standard UML profiles provide, among other features, the ability to define performance annotations: the *UML Profile for Schedulability, Performance and Time* (SPT) defined for UML 1.X versions [OMG 05] and the *UML Profile for Modeling and Analysis of Real-Time and Embedded systems* (MARTE) defined for UML2.X versions [OMG 09].

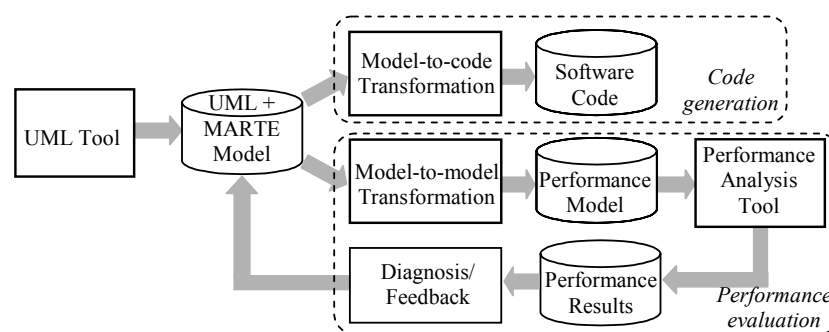


Figure 6.1. MDE model transformations for code generation and performance analysis

Software Performance Engineering (SPE) is a methodology introduced by [SMI 90] with the aim to ensure that software products are built to meet their performance requirements. SPE uses predictive performance models to evaluate the temporal responsiveness of the system (such as response times, delays and throughputs) and to compare architectural design and alternatives for systems with timing and capacity requirements. SPE begins early in the software lifecycle, before serious barriers to performance are frozen into the design and implementation.

Since the introduction of SPE, there has been a significant effort to integrate performance analysis into the software development process throughout all lifecycle phases. A good survey of the techniques for deriving performance models from software specifications is given in [BAL 04].

In the traditional SPE, performance models are built by hand by the analyst. The emergence of model-driven engineering has triggered research in the automatic transformation of software models into performance models. A high degree of automation in building performance models and interpreting their results brings the following benefits:

- higher consistency between the design specification and performance model;
- traceability of performance effects back to design elements and decisions;
- fast performance model update and re-evaluation after a design change;
- stronger analysis for recommending design changes;
- means to bridge the gap between the designer and the performance engineer.

6.2. Performance models

There are two kinds of approaches for analyzing the timing properties of real-time systems: performance and schedulability analysis. Their purpose is different: the first is concerned with estimating average resource capacity, queuing delays due to contention for resources, throughput and identifying bottlenecks, while the second is concerned with finding a feasible schedule that guarantees deadlines inherent to hard real-time systems.

Performance analysis is applied to best-effort and soft real-time systems, such as information processing systems, web-based applications and services, multimedia, telecommunications and enterprise systems. The input parameters and performance results are stochastic variables/processes. On the other hand, schedulability analysis is applied to hard real-time systems with strict deadlines, such as embedded systems, and the analysis is often based on worst-case execution time and deterministic assumptions. This chapter focuses on performance analysis.

It is worth observing that both performance and schedulability models represent the system at runtime, so the models must include not only the performance characteristics of the software itself, but also of the underlying platforms (operating system, middleware, hardware).

A performance model is an abstract representation of a real system that captures its performance properties – mostly related to the quantitative use of resources during runtime behavior – and is capable of reproducing its performance. The model can be used to study the performance impact of different design and/or configuration alternatives under different workloads, leading to advice for improving the system. Performance evaluation of a model may be done either by solving a set of equations by some

analytical (possibly numerical) methods or by simulating the model and collecting statistical results.

Analytical performance models are usually based on underlying stochastic models, which are often assumed to be Markov processes. A Markov process is a stochastic process with discrete state space, where all information about the future evolution of the process is contained in the present state, and not on the path followed to reach this state. Markov models suffer from a problem known as *state space explosion*, whereby its number of states grows combinatorially with the performance model size. This may introduce severe limitations in the size of performance models that can be solved.

Examples of well-known analytical performance models are queuing networks, stochastic Petri nets, stochastic automata networks and stochastic process algebra.

Queuing Network (QN), one of the best known performance models, captures the contention for resources [LAZ 84] very well. Efficient analytical solutions exist for a class of QN (*separable* or *product-form* QN), which make it possible to derive steady-state performance measures without resorting to building the underlying state space. The advantage is that the solution is faster and larger models can be solved. The disadvantage consists in restrictions on model assumptions (e.g., service time distributions, arrival process, scheduling policies). Similar to the approach for product-form QN, approximate solutions have been developed for non-separable QN. There are many extensions to QN in literature. One of them, Layered Queuing Networks, will be discussed in section 6.2.2.

Stochastic Petri Nets (SPN) [AJM 95] are very good flow models able to represent concurrency, but are not as good at representing resource contention and especially queuing policies. Efficient solutions exist only for a limited class of

SPN; most interesting models are solved with Markov chain-based solutions.

Stochastic Automata Networks [PLA 91] are composed of modular communicating automata synchronized by shared events and executing actions with random execution times. The main disadvantage is the state space explosion of its Markovian solution.

Stochastic Process Algebra, introduced in [HIL 94], takes a compositional approach by decomposing the system into smaller subsystems easier to model. This approach is based on an enhanced process algebra, Performance Evaluation Process Algebra (PEPA). The compositional nature of the language provides benefits for model solution as well as model construction. The solution is based on the underlying Markov process.

The target performance model in this chapter is LQN, a QN extension. The following discussion focuses on QN and LQN.

6.2.1. *Queuing network models*

A QN model is a directed graph, whose nodes are service centers, each representing a resource in the system and arcs with associated routing probabilities (or visit ratios) that determine the paths that customers take through the network. Customers representing jobs are flowing through the system, competing for resources.

QN are used to model systems with stochastic characteristics. A QN may have more than one customer class. Each class contains statistical identical customers and has its own workload intensity, service demands and visit ratios. The workload of a customer class may be open (customers arrive with a certain rate, spend time in the

system being served, then leave the system) or closed (the number of customer is fixed; after completing a cycle, a customer starts again). The performance results will be obtained by customer class.

A single service center containing a server and a queue has the following characteristics (represented by Kendall's notation $A/S/c/m/N$):

A = arrival process (e.g. M-Markov, G-general, D-deterministic distribution);

S = service rate (uses distribution identifiers as above);

c = number of servers available serve the customers from the queue;

m = capacity of the queue (infinite by default);

N = customer population (also infinite by default);

scheduling policy (FIFO, LIFO, PS, preemptive priority, etc.).

An important characteristic of QN models is that the functions expressing the queue length and waiting time at a server with respect to workload intensity are very non-linear. An intuitive explanation is as follows: at low workload intensity, an arriving customer meets low competition, so its residence time is roughly equal to its service demand; as the workload intensity rises, congestion increases, and the residence time along with it; as the service center approaches saturation, small increases in arrival rate result in dramatic increases in residence time [LAZ 84].

The non-linearity of performance results makes it difficult to estimate the system performance by simple "rules of thumb", without solving the system of non-linear equations with QN solvers.

Another important concept in a QN is the bottleneck service center, the one that saturates first and throttles the system. Identifying the bottleneck center correctly is important for performance analysis, because the bottleneck must be relieved first in order to improve the system performance. In the case of multiple customer classes, the bottleneck may be different for each class.

QN are widely used for modeling a variety of systems. Although they represent a system at a rather abstract level, QN are a useful tool for predicting the performance of a system. The expected accuracy of QN models according to experience is within 5% to 10% for utilizations and throughputs and within 10% to 30% for response times [LAZ 84].

6.2.2. Layered queuing network model

The LQN model [WOO 95] is a QN extension which can represent nested services (i.e. a server may also be also a client to other servers). A LQN model is a graph whose nodes are either software tasks (thick rectangles) or hardware devices (circles) and the arcs denote service requests, as illustrated in Figure 6.2. The nodes with outgoing but no incoming arcs play the role of clients, the intermediate nodes with both incoming and outgoing arcs, are usually software servers and the leaf nodes are hardware servers. A software or hardware server node can be either a single-server or a multi-server. Software tasks have entries corresponding to different services. Although not explicitly shown in the LQN notation, every server (software or hardware) has an implicit message queue where incoming requests for any offered service are waiting their turn. There are three types of service requests: synchronous (filled arrow), asynchronous (stick arrow) and forwarding (dotted arrow).

Figure 6.2 shows an example of an LQN model of a web server: at the top there are two customer classes with a given number of stochastically identical clients. Each client sends demands for two services e3 and e4 of the WebServer (drawn as thin rectangles attached to the respective task). Every entry has its own execution times and demands for other services (given as model parameters). In this case, the WebServer entries require a service from eCommServer, which in turn calls different entries of two database tasks – a secure and a regular one. Each software task is running on a processor shown as a circle. Also shown as circles are the communication network delays and the disks used by the databases.

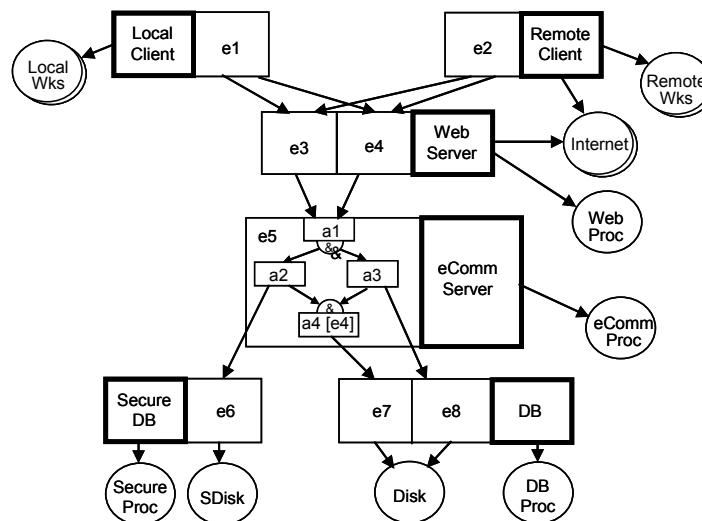


Figure 6.2. Example of LQN model

All arcs used in this example represent synchronous requests, for which the sender is blocked until it receives a reply from the service provider. It is possible to also have asynchronous requests, where the request sender does not expect any reply from the server. Another communication

style in LQN, called “forwarding”, allows a client request to be processed by a chain of servers: the first server in the chain will forward the request to the second, etc., and the last server in the chain will reply to the client.

A server entry may be broken down into two or more sequential phases of service. Phase 1 is the portion of service when the client is blocked waiting for a reply from the server (it is assumed that the client has made a synchronous request). At the end of phase 1, the server will reply to the client, which will unblock and continue its execution. The remaining phases, if any, will be executed in parallel with the client.

An extension to LQN [FRA 00] allows for an entry to be decomposed into activities if more details are required to describe its execution (as for example entry *e5* of task *eCommServer* in Figure 6.2). The activities are connected together to form a directed graph, which may branch into parallel threads of control like in Figure 6.2, or may choose randomly between different branches. Just like phases, activities have execution time demands, and can make service requests to other entries.

6.3. Software model with performance annotations

6.3.1. *Performance domain model*

In order to understand what kind of performance annotations need to be added to UML software models, we need to look at the basic concepts – or in other words at the domain model – for performance analysis. Performance is determined by how the system behavior uses system resources. Scenarios define execution paths with externally visible end points. Quality of Service (QoS) requirements (such as response time, throughput, probability of meeting deadlines, etc.) can be placed on scenarios. In SPT, the

performance domain model describes three main types of concepts: resources, scenarios and workloads [OMG 05].

The resources used by the software can be active or passive, logical or physical software or hardware. Some of these resources belong to the software itself (e.g. critical section, software server, lock, buffer), others to the underlying platforms (e.g. process, thread, processor, disk, communication network).

Each scenario is composed from scenario steps joined by predecessor-successor relationships, which may include fork/join, branch/merge and loops. A step may represent an elementary operation or a whole sub-scenario. Quantitative resource demands for each step must be given in the performance annotations. Each scenario is executed by a workload, which may be open (i.e. requests arriving in some predetermined pattern) or closed (a given number of users or jobs).

In the SPT profile, the domain models for schedulability and performance and their corresponding sub-profiles were defined independently, which made it difficult to reuse annotated models for different analyses. In MARTE (OMG, 2009), the foundation concepts and non-functional properties (NFPs) shared by different quantitative analysis domains are joined in a single package called Generic Quantitative Analysis Model (GQAM), which is further specialized by the domain models for schedulability (SAM) and performance (PAM). Other domains for quantitative analyses, such as reliability, availability and safety, are currently being defined by specializing GQAM.

Core GQAM concepts describe how the system behavior uses resources over time, and contains the same three main categories of concepts presented at the beginning of the section: resources, behavior and workloads.

GQAM Resource Concepts. A resource is based on the abstract *Resource* class defined in the General Resource Model and contains common features such as scheduling discipline, multiplicity, services, etc. The following types of resources are important in GQAM:

- a) *ExecutionHost*: a processor or other computing device on which processes are running.
- b) *CommunicationsHost*: hardware link between devices.
- c) *SchedulableResource*: a software resource managed by the operating system, like a process or thread pool.
- d) *CommunicationChannel*: a middleware or protocol layer that conveys messages.

Services are provided by resources and by subsystems. A subsystem service associated with an interface operation provided by a component may be identified as a *RequestedService*, which is in turn a subtype of *Step*, and may be refined by a *BehaviorScenario*.

GQAM Behavior/Scenario Concepts. The class *BehaviorScenario* describes a behavior triggered by an event, composed of *Steps* related by predecessor-successor relationships. A specialized step, *CommunicationStep*, defines the conveyance of a message. Resource usage is attached to behavior in different ways: a) a *Step* implicitly uses a *SchedulableResource* (process, thread or task); b) each primitive *Step* executes on a host processor; c) specialized steps, *AcquireStep* or *ReleaseStep*, explicitly acquire or release a *Resource*; and d) *BehaviorScenarios* and *Steps* may use other kinds of resources, so *BehaviorScenario* inherits from *ResourceUsage* which links resources with concrete usage demands.

GQAM Workload Concepts. Different workloads correspond to different operating modes, such as takeoff, in-flight and landing of an aircraft or peak-load and average-

load of an enterprise application. A workload is represented by a stream of triggering events, *WorkloadEvent*, generated in one of the following ways:

- a) by a timed event (e.g. a periodic stream with jitter);
- b) by a given arrival pattern (periodic, aperiodic, sporadic, burst, irregular, open, closed);
- c) by a generating mechanism named *WorkloadGenerator*;
- d) from a trace of events stored in a file.

As mentioned above, the Performance Analysis Model (PAM) specializes the GQAM domain model. It is important to mention that only a few new concepts were defined in PAM, while most of the concepts are reused from GQAM.

PAM specializes a *Step* to include more kinds of operation demands during a step. For instance, it allows for a non-synchronizing parallel operation, which is forked but never joins (*noSync* property). A new step subtype, *PassResource*, indicates the passing of a shared resource from one process to another.

In terms of Resources, PAM reuses *ExecutionHost* as its processor, *Schedulable Resources* for processes (or threads) and adds a *LogicalResource* defined by the software (such as semaphore, lock, buffer pool, critical section). A runtime object instance (*PaRunTInstance*) is an alias for a process or thread pool identified in behavior specifications by other entities (such as lifelines and swimlanes).

A UML model intended for performance analysis should contain a structural view representing the software architecture at the granularity level of concurrent runtime components and their allocation to hardware resources, as well as a behavioral view showing representative scenarios with their respective resource usage and workloads.

6.3.2. Source model example

This section presents an example of UML+MARTE source model based on TPC-W, a benchmark of the Transaction Processing Performance Council which models the workload of an on-line bookstore [TPC 02].

The components of TPC-W are logically divided into three tiers: a) a set of emulated web browsers (EB), b) a web tier including web servers and image servers and c) a persistent storage tier. TPC-W emulates customers browsing and buying products from a website, with 14 different web pages that correspond to typical customer operations. The user starts at the “Home” page that includes the company logo, promotional items and navigation options to bestselling books, new books, search pages, the shopping cart, and order status pages. At every page, the user is offered a selection of pages that can be visited next. The user may browse pages containing product information, perform searches with different keys and put items in the cart.

A new customer has to fill out a customer registration page; for returning customers, the personal information is retrieved from the database. Before ordering, the user may update the shopping cart content. When deciding to buy, the user enters the credit card information and submits the order. The system obtains credit card authorization from a Payment Gateway Emulator (PGE) and presents the user with an order confirmation page. At a later date the user can view the status of the last order.

The UML+MARTE source model to be transformed in a performance model is shown in Figure 6.3. It is composed of a structural view showing the concurrent runtime component instances and their deployment to processors in Figure 6.3a, and a behavioral view showing the scenario for one of the pages needed for buying products in Figure 6.3b. Usually the source model contains several performance-

critical scenarios that are used to generate the system performance model, but only one is given here due to space limitations.

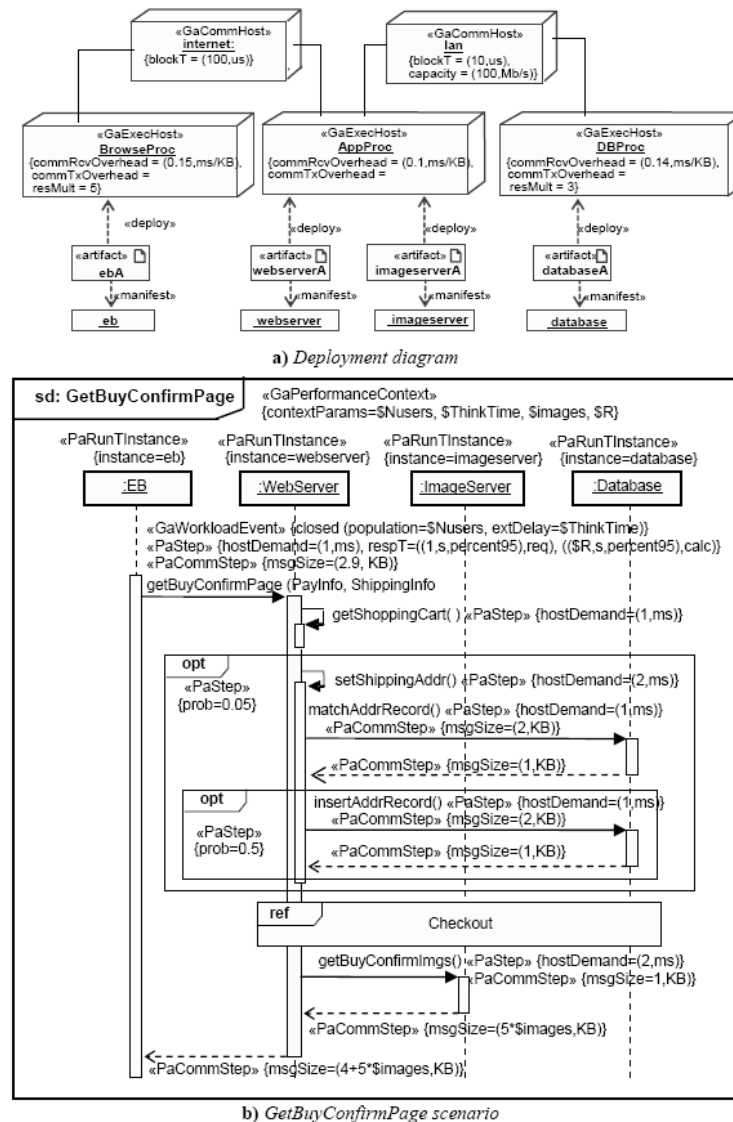


Figure 6.3. Target UML model with MARTE performance annotations

The deployment diagram from Figure 6.3a shows the runtime components at the bottom, their corresponding artefacts and the deployment on processing nodes. The processing nodes are stereotyped as «*GaExecHost*» and the communication network nodes as «*GaCommHost*». The stereotype attributes *commRcvOvh* and *commTxOvh* are host-specific costs of receiving and sending messages, *resMult=5* describes a symmetric multiprocessor with 5 processors, while *blockT* and *capacity* describe a pure latency and bandwidth for the link.

The scenario *GetBuyConfirmPage* is represented in Figure 6.3b. The scenario transfers the shopping cart content into a newly created order for the registered customer, executes a payment authorization, and returns a page with the details of the order to the *EB*. The following operations are performed:

- *EB* issues a request to *WebServer* for “buy confirm page”;
- *WebServer* gets the corresponding shopping cart object;
- with 5% probability (modeled as an **opt** fragment), a shipping address is obtained and *WebServer* tries to match it with information from the database;
- if no address record is found, insert a new address record (modeled as a nested **opt** fragment);
- invoking the *Checkout* sub-scenario (modeled as a **ref** fragment, not shown);
- *WebServer* gets necessary images from *ImageServer*;
- *WebServer* constructs the html code for “buy confirm page” and returns it to *EB*.

Some examples of MARTE performance annotations used in the scenario model are used to indicate the scenario steps, the workload and the concurrent runtime instances

corresponding to the lifeline roles. Two kinds of step stereotypes are applied to messages: «*PaStep*» representing the execution of the operations invoked by the message and «*PaCommStep*» for the communication costs involved with passing the message. Examples of execution step attributes are *hostDemand* giving the value and unit for the required execution time and *prob* giving the probability for the optional steps. The communication steps have an attribute *msgSize* giving the value and unit of the message size. The first step of the scenario has the scenario workload «*GaWorloadEvent*» attached to it, which defines a closed workload with a population given by the variable *\$Nusers* and a think time for each user given by the variable *\$ThinkTime*. Each lifeline role is related to a runtime concurrent component instance, as indicated by «*PaRunTInstance*».

6.4. Mapping from software to performance model

The definition of UML performance annotations has enabled research to transform UML design specifications into many kinds of performance models, based for example on Queueing Networks [COR 00], Layered Queueing Networks [PET 02], [PET 05], [WOO 05], Stochastic Petri nets [BER 02], PEPA [CAV 04], and simulation [BAL 03].

In this section, the mapping concepts from software to performance models are explained using a direct transformation from annotated UML to LQN; another possible transformation approach using a pivot language is discussed in section 6.5. In the direct approach, the structure of the LQN model is generated from the high-level software architecture and deployment. In principle, active software component instances and hardware devices (which are all resources) are mapped to LQN tasks. In some cases, LQN tasks are also generated from passive instances, which are logical resources shared by active instances. In fact, the

mapping to tasks is guided by the architectural patterns used in the system, such as pipeline and filters, client/server, client/broker/server, layers, master-slave, blackboard, etc. Each pattern describes two inter-related aspects: its structure (what are the interacting components) and behavior (how they interact). The architectural pattern components are usually concurrent entities that execute in different threads of control, compete for resources, and may require some synchronization in their interaction. For more details on the transformation rules from UML to LQN based on different architectural patterns see [PET 00].

Figure 6.4 gives the direct transformation algorithm from annotated UML to LQN, assuming that the scenario models are represented by sequence diagrams. A similar pattern-based approach is presented in [PET 02], where the scenarios are modeled as activity diagrams. A graph-grammar-based algorithm was proposed to divide the activity diagram into activity subgraphs, which are further mapped to LQN phases or activities. Such a transformation from software to performance model is an *abstraction-raising transformation*, as shown in [PET 05].

1. Generate LQN model structure
 - 1.1 map high-level component instances to LQN tasks according to patterns;
 - 1.2 map deployment diagram nodes to LQN hardware devices;
2. Generate LQN entries, phases, activities from scenarios
 - 2.1 for each scenario {
 - 2.1.1 generate a LQN reference task and its dummy processor corresponding to the scenario workload;
 - 2.1.1 match messages with inter-component communication style from patterns;
 - 2.1.2 map external message calls to entries;
 - 2.1.3 for each entry {
 - 2.1.3.1 group corresponding execution occurrences according to patterns;
 - 2.1.3.2 map groups to phases or activities;
 - 2.1.3.3 for each phase and activity
 - 2.1.3.3.1 compute service time and number of calls;

Figure 6.4. Algorithm for direct transformation from annotated UML to LQN

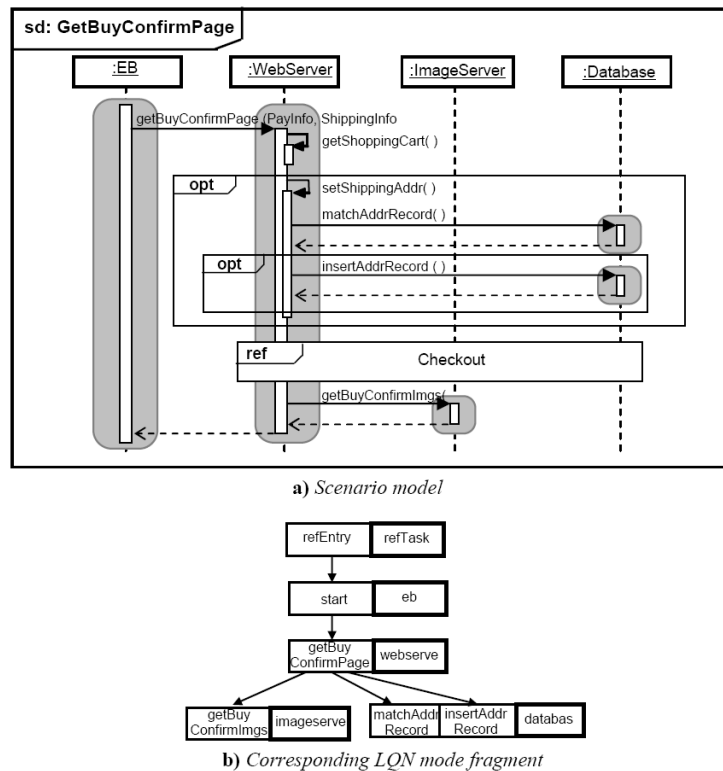


Figure 6.5. Mapping between the scenario model and the performance model

Figure 6.5 illustrates the application of the algorithm from Figure 6.4 (more specifically, the loop body 2.1.1 to 2.1.3) to the scenario *GetBuyConfirmPage* from Figure 6.3. It so happens that a single architectural pattern – client/server – is used repeatedly in this scenario, as the four lifeline roles interact through synchronous messages. The corresponding LQN model fragment shown in Figure 6.5b contains five LQN tasks: four correspond to the active runtime component instances *eb*, *webserver*, *imageserver* and *database* (according to the «*PaRunTInstance*» stereotypes from Figure 3.b) and the fifth, *refTask*, is a reference task controlling the scenario workload. Each task has an entry for every external message it receives. For example, the database task has two entries

because two different calls, *matchAddrRecord* and *insertAddrRecord* are made to this task. The shaded areas in Figure 6.5.a are grouping behavior occurrences (stereotyped as scenario steps) which are further mapped to phases belonging to entries. (This example has no LQN activities). For each entry, the group of steps executed between the acceptance of the corresponding synchronous request and the sending of the reply is mapped to phase 1. For instance, the steps corresponding to the behavior executions triggered by the messages *getBuyConfirmPage*, *getShoppingCart* and *setShippingAddr* are all included in phase 1 of entry *getBuyConfirmPage*. Also included in this phase are the steps executed by this lifeline inside the fragment *Checkout* (which is not detailed here). In fact, the fragment *Checkout* may also add entries to the tasks *imageserver* and *database* corresponding to all the messages sent to these lifelines by other lifelines. The service time parameter of each phase is obtained by summing up the host demand of the included steps. The number of calls made by every phase to other entries is similarly obtained.

6.5. Using a pivot language: Core Scenario Model (CSM)

A *pivot language*, also known as *intermediate* or *bridge* language, can be used as an intermediary for translation in cases where many source languages are translated to many target languages. A pivot language avoids the combinatorial explosion of translators across every combination of languages and allows for a smaller semantic gap during each transformation. Such an approach is taken, for example, in the model-driven performance evaluation project called Performance by Unified Model Analysis PUMA [WOO 05] which enables the integration of performance analysis in a UML-based software development process. PUMA uses a pivot language Core Scenario Model (CSM) to extract and audit performance information from different kinds of design

models (e.g., different UML versions of activity and sequence diagrams) and to support generation of different kinds of performance models (e.g. QN, LQN, Petri nets, simulation). Figure 6.6 illustrates the PUMA transformation and analysis chain. There are other intermediate languages for performance analysis proposed in literature, such as Klaper [GRA 05] and Palladio Component Model [BEC 07].

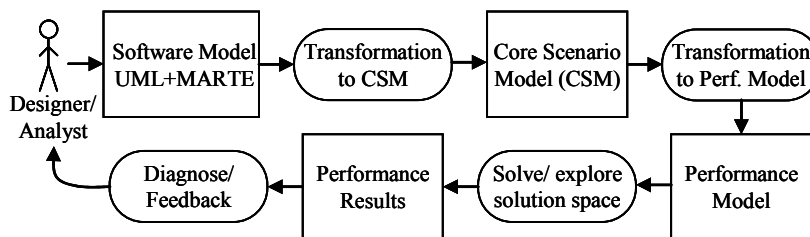


Figure 6.6. *PUMA transformation and performance analysis chain*

CSM is focused on modeling scenarios, which are implicit in many software specifications; they are useful for communicating partial behaviors among diverse stakeholders and provide the basis for defining performance characteristics.

The CSM metamodel is similar to the SPT Performance Profile, describing three main types of concepts: resources, scenarios and workloads. A scenario is a graph of steps with precedence relationships. A step may represent a basic operation or be refined as a sub-scenario. There are the following kinds of resources in CSM: a) *ProcessingResource* – a node in a deployment diagram; b) *ComponentResource* – a process or active instance related to a lifeline role in a sequence diagram or a swimlane in an activity diagram; c) *LogicalResource*; and d) *external resource* – a resource not explicitly represented in the UML model required for executing external operations that have a performance impact (for example, a disk operation).

6.6. Case study performance model

The performance experiments conducted with the LQN model of the TPC-W scenario from Figure 6.3 compares two design alternatives: one for the source model as presented in section 6.3.2 and the other after adding SSL secure communication between the user browser and the webserver. Both performance models include the LQN model elements generated from the *Checkout* fragment (not given in this chapter). Details on how to add security enhancements to a system model in general and to the TPC-W in particular can be found in [WOO 09] and [HOU 10].

Figure 6.7 shows the simplified LQN models (only the tasks and devices) for the two alternatives without and with SSL. The shaded tasks from Figure 6.7b have been added to perform the SSL functionality (encryption and decryption being the most important functions) on the user and webserver side. The dotted arrows represent forwarding requests.

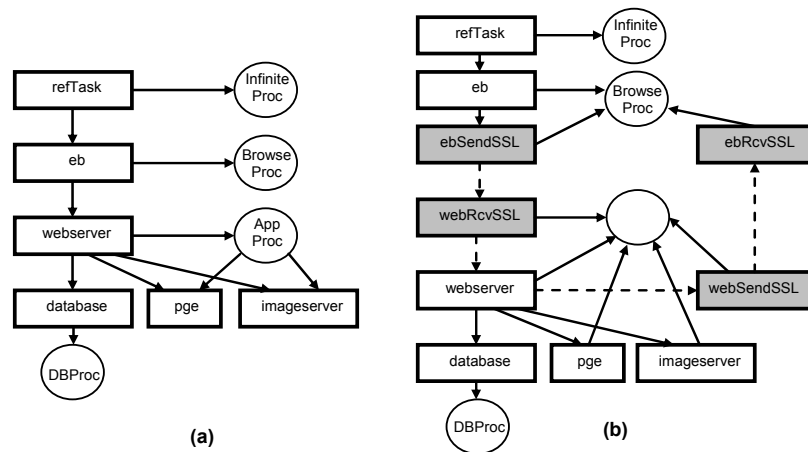


Figure 6.7. LQN model for the example system:
(a) without SSL; (b) with SSL

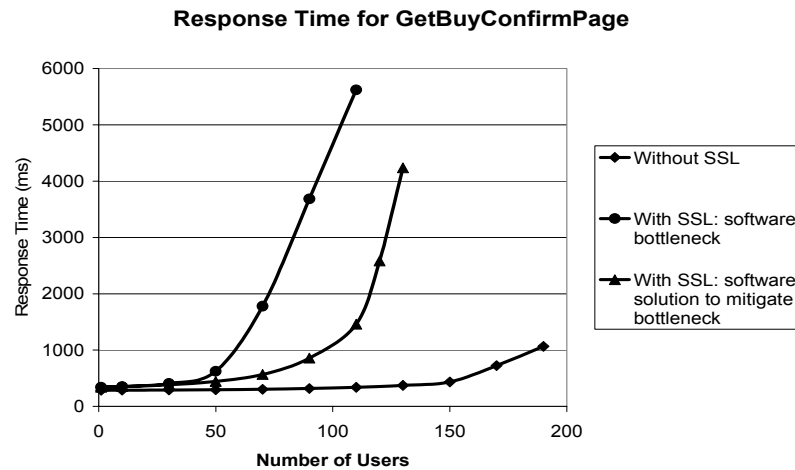


Figure 6.8. Response time for *GetBuyConfirmPage* scenario without and with SSL

The results of the LQN experiments for the *GetBuyConfirmPage* scenario are shown in Figure 6.8. The three curves represent the response time versus the number of users for the following design/configuration alternatives:

- a) The lowest curve corresponds to the initial model of the scenario without SSL. The concurrency level of the software tasks has been chosen such that the system gives the maximum performance for the given hardware configuration.
- b) The highest curve corresponds to the model with SSL, for the concurrency level obtained immediately after adding SSL, without any attempt to optimize for performance. The response time has a typical non-linear shape with a “knee” around 60, after which it grows very fast due to the saturation of the system.
- c) The middle curve corresponds to the SSL enhanced system with an improved software configuration. The problem before this improvement was that one of the software tasks charged with security functions on the server side becomes saturated, even though the hardware resources

are not used at maximum capacity. Such a situation is known as a “software bottleneck”. The solution is to increase the concurrency level, in this case to introduce more threads for the bottleneck task, in order to use the available capacity of the hardware resources. The response time improves and the new bottleneck moves to the processor running the *webserver*. The next performance solution would need to add new processing capacity at the hardware resource level.

6.7. Conclusions

Experience in conducting model-driven NFP analysis in the context of MDE shows that the domain is still facing a number of challenges.

Human qualifications

Software developers are not trained in all the formalisms used for the analysis of different non-functional properties (NFPs), which leads to the idea that we need to hide the analysis details from developers. However, the software models have to be annotated with extra information for each NFP and the analysis results have to be interpreted in order to improve the designs. A better balance needs to be made between what is to be hidden and what is to be exposed.

Abstraction level

The analysis of different NFPs may require source models at different levels of abstraction/detail. The challenge is to keep all the models consistent.

Tool interoperability

Experience shows that it is difficult to interface and to seamlessly integrate different tools, which were created at

different times with different purposes and may be running on different platforms.

Software process

Integrating the analysis of different NFP raises process issues. For each NFP it is necessary to explore the state space for different design alternatives, configurations and workload parameters in order to diagnose problems and decide on improvement solutions. The challenge is how to compare different solution alternatives that may improve some NFPs and deteriorate others, and how to decide on trade-offs.

Change propagation through the model chain

Currently, every time the software design changes, a new analysis model is derived in order to redo the analysis. The challenge is to develop incremental transformation methods for keeping different models consistent instead of starting from scratch after every model improvement.

6.8. Acknowledgements

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), through its Discovery and Strategic Projects programs.

6.9. Bibliography

- [AJM 95] AJMONE MARSAN M., BALBO G., CONTE G., DONATELLI S. and FRANCESCHINIS G., *Modelling with Generalized Stochastic Petri Nets*, Wiley Series in Parallel Computing, John Wiley and Sons, 1995.

- [BAL 03] BALSAMO S., MARZOLLA M., "Simulation Modeling of UML Software Architectures", *Proc. ESM'03*, Nottingham (UK), June 2003.
- [BAL 04] BALSAMO S., DI MARCO A., INVERARDI P., SIMEONI M., "Model-based performance prediction in software development: a survey", *IEEE Transactions on Software Engineering*, Vol 30, No.5, pp.295-310, May 2004.
- [BEC 07] BECKER S., KOZIOLEK H., REUSSNER R., "Model Based Performance Prediction with the Palladio Component Model", *Proceedings of the 6th ACM Int. Workshop on Software and Performance*, Buenos Aires, Argentina, February 2007, p.54-65.
- [BER 02] BERNARDI S., DONATELLI S., MERSEGUER J., "From UML sequence diagrams and statecharts to analysable Petri net models", in *Proc. 3rd Int. Workshop on Software and Performance*, Rome, July 2002, pp. 35-45.
- [CAV 04] CAVENET C., GILMORE S., HILLSTON J., KLOUL L., and STEVENS P., "Analysing UML 2.0 activity diagrams in the software performance engineering process", in *Proc. 4th Int. Workshop on Software and Performance*, Redwood City, CA, January 2004, pp. 74-83.
- [COR 00] CORTELLESA V., MIRANDOLA R., "Deriving a Queueing Network based Performance Model from UML Diagrams", in *Proc. Second Int. Workshop on Software and Performance*, Ottawa, Canada, September 17-20, 2000, pp. 58-70.
- [FRA 00] FRANKS G., Performance Analysis of Distributed Server Systems, PhD Thesis, Carleton University, Systems and Computer Engineering, Report OCIEE-00-01, Jan. 2000.
- [GRA 05] GRASSI V., MIRANDOLA R., SABETTA A., "From design to analysis models: a kernel language for performance and reliability analysis of component-based systems", *Proceedings of the 5th Int. Workshop on Software and Performance*, Palma, Spain, July 2005, p.25-36.
- [HIL 96] HILLSTON J., *A Compositional Approach to Performance Modelling*, Cambridge University Press, 1996.

- [HOU 09] HOUMB S.H., GEORG G., PETRIU D.C., BORDBAR B., RAY I., ANASTASAKIS K., and FRANCE R.B., “Balancing Security and Performance Properties During System Architectural Design”, in *Software Engineering for Secure Systems: Industrial and Research Perspectives*, H.Mouratidis (Ed), IGI Global, 2009.
- [LAZ 84] LAZOWSKA E., ZAHORJAN J., SCOTT GRAHAM G., SEVCIK K.S., *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, 1984.
- [OMG 05] Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification (SPT)*, Version 1.1, OMG document formal/05-01-02, January 2005.
- [OMG 09] Object Management Group, *A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems)*, Version 1.0, OMG doc. formal/2009-11-02, December 2009.
- [PET 00] PETRIU D.C., SHOUSA C., JALNAPURKAR A., “Architecture-Based Performance Analysis Applied to a Telecommunication System”, *IEEE Transactions on Software Engineering*, Vol.26, No.11, pp.1049-1065, November 2000.
- [PET 02] PETRIU D.C., SHEN H., “Applying the UML Performance Profile: Graph Grammar-based derivation of LQN models from UML specifications” in *Computer Performance Evaluation - Modelling Techniques and Tools*, (Tony Fields, Peter Harrison, Jeremy Bradley, Uli Harder, Eds.) LNCS Vol. 2324, pp.159-177, Springer, 2002.
- [PET 05] PETRIU D.C., “Performance Analysis with the SPT Profile”, in *Model-Driven Engineering for Distributed and Embedded Systems* (J. Champeau, J.P. Babau, S. Gerard, eds.), pp. 205-224, Hermes Science Publishing Ltd., London, England, 2005.
- [PET 06] PETRIU D.C., SABETTA, A. “From UML to Performance Analysis Models by Abstraction-raising Transformation”, in *From MDD Concepts to Experiments and Illustrations*, (eds. J.P. Babau J-P., Champeau J., Gerard S.), ISTE Ltd., pp.53-70, 2006.

- [PET 07] PETRIU D.B., WOODSIDE C.M., “An intermediate metamodel with scenarios and resources for generating performance models from UML designs”, *Software and Systems Modeling*, Vol.6, No.2, pp. 163-184, June 2007.
- [PLA 91] PLATEAU B., ATIF K., “Stochastic Automata Network of Modeling Parallel Systems”, *IEEE Transactions on Software Engineering*, Vol.17, No.10, p.1093-1108, 1991.
- [SMI 90] SMITH C.U., *Performance Engineering of Software Systems*, Addison-Wesley Publishing Co., New York, NY, 1990.
- [TPC 02] Transaction Processing Council, TPC Benchmark W (Web Commerce) Specification, Version 1.8, February 19, 2002.
- [WOO 95] WOODSIDE, J.E. NEILSON, D.C. PETRIU, S. MAJUMDAR, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software”, *IEEE Trans. on Computers*, Vol.44, No.1, pp. 20-34, January 1995.
- [WOO 05] WOODSIDE C.M., PETRIU D.C., PETRIU D.B., SHEN H, ISRAR T., MERSEGUER J., “Performance by Unified Model Analysis (PUMA)”, *Proc. 5th Int. Workshop on Software and Performance WOSP'2005*, pp. 1-12, Palma, Spain, 2005.
- [WOO 09] WOODSIDE C.M., PETRIU D.C., PETRIU D.B., XU J., ISRAR T., GEORG G., FRANCE R., BIEMAN J., HOUMB S.H., JÜRJENS J., “Performance Analysis of Security Aspects by Weaving Scenarios from UML Models”, *Journal of Systems and Software*, vol.82, p.56–74, 2009.

Chapter 7

Model Integration for Formal Qualification of Timing-Aware Software Data Acquisition Components

7.1. Introduction

Computers are increasingly used for process control systems (transport systems, industrial processes). The role of these computers is to implement control laws using data (giving information about the process state) coming from sensors and producing outputs (command) to control a process using actuators. Due to the dynamic evolution of the controlled process, a control law implementation needs to satisfy some timing constraints (data arrival law, data lost rate, time interval between data update and command production) [WIT 95], [SWT 05], [TOR 98], [FEN 01]. Because most of these systems are critical systems, timing constraints are to be *a priori* (before execution) formally validated.

Chapter written by Jean-Philippe BABAU, Philippe DHAUSSY and Pierre-Yves PILLAIN.

Critical constraints lead to predictable systems that induce hand-made and code-centric developments. Nowadays, these systems are increasingly complex and reuse of software components will both help the designer and reduce development cost. In this domain, because of critical timing constraints, reusable components must be timing-aware components.

In the process control domain, reusable software components are classically execution services provided by a Real-Time Operating System respecting standards such as OSEK [OSEK] or POSIX [POSIX] remote communication services (FT layer of TTA [SCH 97]) and device drivers. In this work, we consider sensor drivers, that is to say the software data acquisition part of process control systems [FOK 02]. The purpose of a software data acquisition driver is to provide a well adapted interface between the physical sensors (which produce data) and the applications (which consume data). For these components, depending on applications, the most relevant timing characteristics can differ [RAM 99] [BAT 03]. Thereafter, in the control process domain, one of the most useful timing characteristic is minimal and maximal delay of produced data, a characteristic we consider in this chapter.

The most common techniques to evaluate maximum delays are based on task and message scheduling analysis [LIU 73], [AUD 93], [TIN 94], [KLE 93], [BER 03], [MIG 02]. These techniques consider simple architectures, generally static, and may give unrealistic bounds. In order to limit these problems, many works has been proposed by using modeling techniques based on exhaustive simulation of timing behaviors such as communicating timed automata or hybrid automata [PET 99], [HIE 03], [LIM 04], [BEL 04], [WAS 08].

Using exhaustive simulation makes it possible to compute more general and realistic timing information for complex

architectures (dynamic behaviors). Based on IF [BOZ 02], previous work [BEN 05] shows the interest of such an approach for characterizing minimal and maximal delays for acquisition drivers. These works show that timing characteristics of an acquisition part strongly depend on internal parameters of acquisition drivers (such as buffer size and communication policies) and also on external stimuli pattern (arrivals law from the sensors, application consumption law). Then, to get a qualification of an acquisition component for reuse, it is necessary to formally describe its contextual use and/or to provide the ability to tune internal parameters to satisfy specific contextual constraints.

The CDL is a recent work [DHA 08], [DHA 09a], [DHA 09b] which provides formal and high-level models to describe the context for concurrent systems. This description is then used to drive exhaustive simulation and to qualify a component, regarding explicit properties, for a certain context.

Based on these previous works, the aim of the paper is to present the use of modeling techniques to qualify, from a timing point of view, data acquisition driver models, expressed using MARTE profile [OMG 09]. A timing observation tool, based on IF observers and formal exhaustive simulation is used to evaluate the delay characteristic. Then, the CDL modeling approach is used to qualify the timing impact of internal parameter choices for different context usage.

The rest of this chapter is organized as follows. Section 7.2 presents data acquisition system modeling illustrated by case studies, IF modeling for concurrent application behavior modeling, CDL models and timing characteristic observers. Section 7.3 presents how to use CDL to describe the parameterization of a data acquisition driver's usage. Then, section 7.4 shows how to use a timing evaluation tool (in

terms of maximal and minimal delays) for different implementations and contexts of a case study. We finally conclude the paper and give some perspectives.

7.2. System modeling

7.2.1. Acquisition system modeling

A data acquisition system (see Figure 7.1) is classically a compound of several elements, divided into hardware (HW) and software (SW) resources:

- *PhysicalSensor* (HW): this converts information (temperature, speed) coming from the environment into numerical data¹.

- *CommunicationInterface* (HW), also called the hardware driver: this connects a physical sensor to the software part of the system. It recovers data produced by the physical sensor (analog-to-digital converter), implements protocols between the sensors and the computing resource (and especially medium access control policies) and stores them in registers accessible by software drivers. So *CommunicationInterface* is allocated both on the sensor and applicative part.

- *DeviceDriver* (SW): this is a dedicated piece of software, usually integrated into an operating system, and is independent of applications. It abstracts the hardware layer and can be used in various applicative contexts [MER 00]. It acts as a logical view for the corresponding HW device, managing device information. For sensors, it receives data, storing them in device buffers, and transmits them to applications, depending on applicative consumption policy.

1. We can note that software may be embedded in smart sensors, but this is beyond the scope of this chapter, so, here, a sensor is viewed as a HW resource.

– *RT_Application* (SW): an RT application regularly reads data from device drivers and uses them to compute some commands. A read operation may be a non-blocking (polling) or a blocking (waiting for new incoming data) operation.

– *RTOS* (SW): a Real-Time Operating System provides an API for concurrent programming based on real-time scheduling algorithms.

Getting a performance evaluation from the device driver component requires formal modeling tools for all the system, for the context and for performance evaluation. Due to concurrent and timing aspects of the system, the modeling tools are communicating timed automata for formal system behavior modeling, timed observers for delay evaluation and CDL for context modeling.

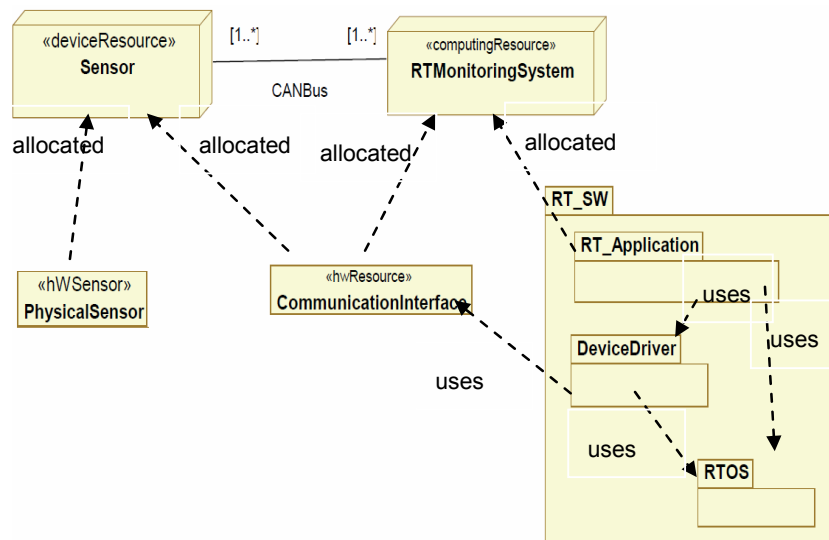


Figure 7.1. MARTE model of acquisition system architecture (Papyrus tool)

7.2.2. Case study

This section presents a case study to illustrate the proposed approach.

Our running example is a multi-task application whose goal is to handle a set of sensing peripherals. The physical part of the system is made of detection sensors communicating using a CAN Network. Each sensor emits its address (coded here using one byte) each time an object is detected. For one sensor, the minimal time between two activations (and so sending two messages) is equal to 100 ms. Having arrived on *RTMonitoringSystem*, *Communication Interface* stores the sensor address using an internal register (named *sensorRegister*) and then triggers an interrupt which starts *ITMessage* handler. This is awoken every time a new message occurrence comes from a sensor. After some processing time (corresponding to the decoding of information read from the sensor buffer) it then stores this occurrence to the *SensorMessages* FIFO queue. This handler execution time is less than the minimal separation time between the arrival of two messages.

For software architecture (see Figure 7.2), the device driver is based on a mono-task, called *DriverTask*, architecture. Arrival of a new piece of data triggers a handler *ITMessage*. The handler reads the incoming data in a specific register and stores it in a FIFO queue *SensorMessages*. *DriverTask* is pending on the queue; after reading a new message, performing a particular data operation (filtering, conditioning, etc.), it stores a new copy of the data in a specific sensor-buffer *dataBuffer* (one buffer *dataSensorI* per sensor). The RT application periodically (using alarm *Activation*) performs a read operation by reading the corresponding sensor-buffer and printing an activated sensor ID on *Screen*. To ensure mutual exclusion

on sensor-buffers, in order to avoid interleaving deadlocks, access to sensor-buffers is protected by a unique mutex (one for all the sensor-buffers). For multitasking modeling, it is important to note that RT applications have less priority than drivers.

All the elements (sensors, CommunicationInterface, Driver task, ApplicationReader and communications objects) are considered to have been created at the beginning of the system. However, activation date of active elements (sensors and tasks) are not *a priori* defined (see variation points).

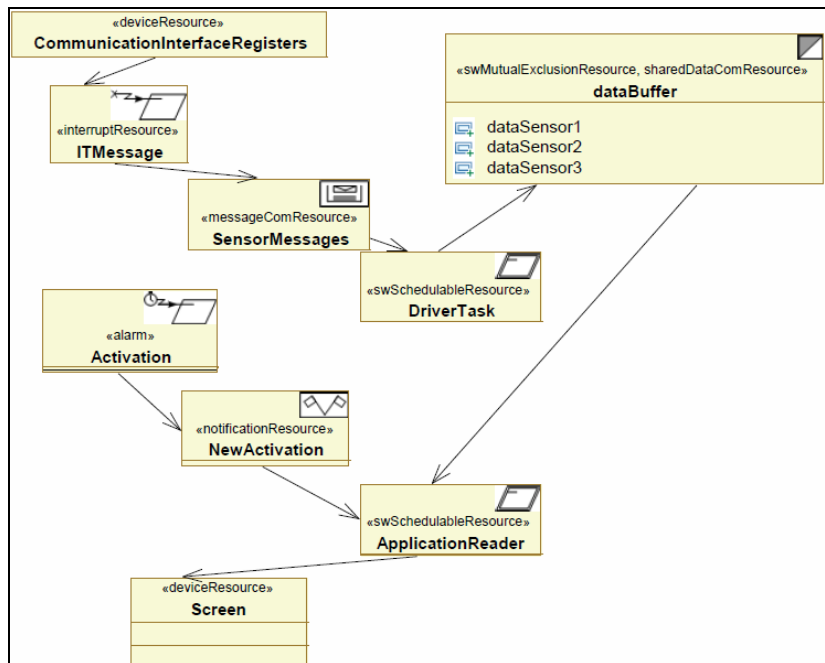


Figure 7.2. MARTE model of software part (*DeviceDriver* and *RT_Application*) (Papyrus tool)

For safety aspects, the system has to consider all the sensor messages (no data lost). Moreover each RTOS entity has to be safely used (correct API usage). As a consequence

of this, we have to *a priori* check that the FIFO mailbox is never full when the ITMessage handler posts a new message on it (no control at runtime). For timing constraints, the printing operation has to consider recent information (age of data is less than 50 ms).

To consider such constraints, the designer has to find the correct internal parameters of the driver to ensure, in a given context (sensor and application behavior), that the system respects both safety and timing constraints. This a problem of architectural property design, guided by performance consideration.

7.2.3. Formal modeling techniques

7.2.3.1. System behavior modeling

First, this section gives the main information about the techniques used for formal modeling of such a system and its timed behavior. The technique is described in [BEL 04] and based on one of the well-known and existing formalisms for timed communicating automata: IF [BOZ 02]. Using IF, a system is described by a set of communicating timed processes [ALU 94]. An IF process is modeled by communicating timed automaton. IF processes communicate themselves asynchronously by sending signals or sharing data.

Each process owns a FIFO queue (with or without loss) to manage incoming signals. From a time execution flow point of view, it is important to note that IF considers either discrete time or dense time (which we consider in this chapter). To avoid timelocks, time progress conditions are associated with the transitions in the form of deadlines [SIF 96], [BOR 97]. Finally, from an IF specification, it is possible to generate a corresponding labeled transitions system

(LTS). From LTS and using specific tools such as IFx² [BOZ 02], it is possible to have an exhaustive simulation of all the timing behavior of the system. To reduce combinatory explosion, it is possible to limit interleaving by defining an order (based on priority levels) between process transitions.

In the end, based on timed automata principles, IF appears as an adapted modeling tool for such a modeling activity: it provides formal semantics. It integrates concurrent data and timing aspects; it provides automatic transformation to simulation and verification tools IFx and OBP *Observer Based Prover*³ [DHA 08].

Following [BEL 04], to describe the timing behavior of a multitasking system, the model of the system is divided into four parts:

- the physical process (sensor in our case) is modeled as triggers;
- the HW part (communication interface) corresponds to registers and interrupt services (interrupt controller);
- the RTOS services are based on communication objects (semaphores, mailboxes, etc.) and interrupt routines or handlers;
- and the applicative tasks (driver and application tasks).

Each physical entity (sensor), the communication interface and each RTOS entity (task, routines, communication objects such as semaphores or mailboxes) is described using an IF process.

2. IFx is IF simulator, developed by VERIMAG Lab.

3. OBP is available (version 2.0) under EPL license at:
<http://gforge.enseiht.fr/projects/obp>.

```

process FifoBox(0);

var nbMsg integer;
var FIFO T_data_FIFO;
var FIFO_Message SENSOR_DATA;
var request boolean;

state initial #start ;

    ... /* initialization phase for counters */
    nextstate ready;
endstate;

state ready ;
priority 4;                                     /* priority of the process */
input sendToMailbox(FIFO_Message);             /* message sent by driver
task process */
if (nbMsg < FIFO_Size) then
    task FIFO[ nbMsg ]:= FIFO_Message;
    task nbMsg := nbMsg + 1;
else
    output ErrorOverflowMailBox();               /* overflow */
    output lessData(FIFO_Message);              /* occurrence is lost */
endif
nextstate ready;

priority 4;
input readFromMailbox();                       /* message sent by
applicative task process */
if (nbMsg = 0) then
    task request := true;                       /* reader task is pending */
else
    output msgFromMailbox(FIFO[ 0 ]) to {Driver}0; /* return to reader task */
    ... /* deleting red message in the FIFO */
endif
nextstate ready;

priority 4;
provided ((request = true) and (nbMsg > 0));    /* a new message is arrived and the
task is pending */
task request := false;
output msgFromMailbox(FIFO[ 0 ]) to {Driver}0;
... /* deleting red message in the FIFO */
nextstate ready;
endstate;
endprocess;

```

Table 7.1. IF model of FIFO mailbox with one possible reader

Signals, exchanged between IF processes, represent either data production by sensors or by communication interfaces (*newValue(data)*), or RTOS primitive calls (*InterruptHanler()*, *createTask()*, *sendToMailbox()*, *semaphoreTake()*, ...). Then, for the SW part, the model is enriched to consider execution time and the scheduling algorithm. The idea is to discretize time for execution (a task is view as a sequence of consecutive units of time) and to order IF process execution following system execution constraints: priority (environment processes) > priority (HW processes) > priority (RTOS objects) > priority (interrupt routines) > priority (tasks process). For each priority class (except for tasks), we assume that entities are independent and so, may be *a priori* ordered: at an instant, execution of two concurrent transitions lead to the same state. This assumption makes some patterns impossible; for instance, two different IT routines cannot send a message in the same mailbox.

For task processes, priorities are fixed according to user priorities, following a fixed priority scheduling, as usual using RTOS. Priorities are associated with each transition, defining a pre-order, reducing combinatory explosion. Then, to reduce combinatory explosion, system phases are considered to produce independent LTS for each distinct phase. In this chapter, we concentrate on the main phase (this phase follows the initialization phase) where all the entities has been correctly initialized.

7.2.3.2. Context description

This section gives the CDL principles to describe several scenarios.

In our approach, CDL aims at formalizing the context with scenarios and temporal properties using property patterns. This DSML⁴ is based on UML 2. A CDL model

4. Domain Specific Modeling Language.

describes, on the one hand, the context using activity and sequence diagrams and, on the other hand, the properties to be checked using property patterns. The originality of CDL is its ability to link each expressed property to a context diagram, i.e. a limited scope of the system behavior. For formal validation, CDL associates formal semantics with UML models, described in term of traces [DHA 09a]. The language is designed and tooled to offer a simple and usable context description framework.

The semantics of the CDL language is specified in multiple and complementary ways. One is the metamodel (e.g. the domain ontology), another is the concrete syntax. The metamodel is an ECore model (EMF). It is annotated with OCL invariants to enforce its semantics. A diagrammatical concrete syntax is created for the context description and a textual syntax for the property expression. The following sections outline: (i) the proof context formalization, (ii) the property expressions.

In [DHA 08], we proposed a context description language using UML 2 diagrams (see Figure 7.2 for the illustration of the case study). It is inspired by *Use Case Charts* of [WHI 05]. We extend this language to allow several entities to compose the proof context. These entities run in parallel. CDL is hierarchically constructed in three levels: Level-1 is a set of diagrams of use cases which describes hierarchical activity diagrams. Either an alternative between several executions (decision/merge) or a parallelization of several executions (fork/join) is available. Level-2 is a set of scenario diagrams organized by alternatives. Each scenario is fully described at Level-3 by UML 2 sequence diagrams. These diagrams are composed of two lifelines, one for the proof context and another for the *model under study* (MUS).

Delayable interaction event occurrences are specified on these lifelines. Counters limit the loops of diagram executions. This ensures the generation of finite context

automata, as described in [DHA 08], [DHA 09b]. Transitions at Level-1 and Level-2 are enabled according to the values of some untimed guards or timed guards. As mentioned in the introduction, the approach links the context description (Level-1 or Level-2) to the specification of the properties to be checked by stereotyped links property/scope. A property can have many scopes and many properties can refer to a single diagram.

Semantics of Level-1 and Level-2 is described in term of traces, inspired by [Hau05]. Level-1 and Level-2 are based on the semantics of the scenarios and expressed by construction rules of sets of traces built using *seq*, *alt* and *par* operators (*par* only for Level-1). At Level-3, the semantics of a scenario is expressed by a set of traces as described in [HAU 05] and in accordance with the semantics of UML 2 sequence diagrams. A scenario trace is an ordered events sequence which describes a history of the interaction between the context and the model. A scenario with several interactions is described by a set of traces.

7.2.3.3. *Timing observers*

This section shows how to formally evaluate properties by using observers.

Safety properties

For the property specifications, we follow a pattern-based approach and integrate property pattern descriptions in the CDL language (we refer the reader to [DHA 09a] for details). The patterns [DWY 99] are classified into basic families, which take into account the timed aspects of the properties to be specified. The patterns identified allow the answer (*Response*), necessity (*Precedence*), absence (*Absence*), and existence (*Existence*) properties to be expressed. The properties refer to detectable events like transmissions or receptions of signals, actions, and model state changes.

These basic forms are enriched by options (*Pre-arity*, *Post-arity*, *Immediacy*, *Precedence*, *Nullity*, *Repeatability*) using annotations [KON 05]. The property must be taken into account during all the model execution, before, after or between occurrences of events. Patterns have the possibility of expressing guards on the occurrences of events expressed in the properties [DHA 09a]. Guards refer to variables declared in the context model. This mechanism gives precision to the property/scope reference introduced in the previous section.

Another extension of the patterns is the possibility of handling sets of events, ordered or not ordered as in the proposal of [JAN 99]. The operators *An* and *All* specify respectively if an event or all the events, ordered (*Ordered*) or not (*Combined*), of an event set are concerned with the property. Illustrating with our case study, Figure 7.3 depicts one *Absence* property *AbsenceOverflow* obtained from the *R1* requirement: “a message is never posted when the mailbox is full”.

With the CDL language, this property is expressed as follows (Table 7.2).

Property AbsenceOverflow
AN
exactly one occurrence of ErrorOverflowMailBox
end
occurs never

Table 7.2. *AbsenceOverflow* property

Our OBP toolset transforms each property into an observer automaton including a *reject* node. With observers, the properties we can handle are of safety and bounded liveness type. The accessibility analysis consists of checking if there is a *reject* state reached by a property observer. For the property (Figure 7.3), a *reject* node is reached after

detecting event *ErrorOverflowMailBox*. Consequently, such a property can be verified by using reachability analysis implemented in a formal model checker.

Timing properties

For timing characteristics, delay in our case, of data flow we use specific performance observers. Informally, a delay is the interval of time between the instant when data changes on the controlled process and the time when an application reads it. Because data is vehicled through a data flow, timing characteristic is related to data flow.

Formal techniques require expression of timing properties in a specific way that makes possible its verification. To express timing properties, we use either real time logic [MAN 92], or specific timed automaton called observers [HAL 93]. Logics and observers respond on property verification in the form: valid property (reachable *success* state for an observer) or invalid property (reachable *error* state for an observer). Logics and observers are well adapted to Boolean property (*success* or *error*) like safety or liveness properties.

A delay may be viewed as a bounded liveness property, classically expressed as: “*after a data occurrence occurs, the driver will always produce an output before x unit of times*”. This approach is only able to check if a delay is more or if it is a less specific value (*a priori* given). Using this approach, getting the exact timing performances of a system, evaluating delay implies to implement a dichotomic analysis of the system. To avoid such a heavy approach, observers (well adapted to bounded liveness properties) have to evaluate all the possible delays in the system. A performance observer is then an instrumented bounded liveness observer.

The followed approach consists of monitoring, using observers, the system catches all occurrence data in the

system between their production and their consumption [MOR 09]. A performance observer *occLifeCycle* (see Table 7.3) is then dedicated to observe the life cycle of one occurrence of the data flow, matching creation occurrence (*InitData*), adding copies of occurrences (*MoreData*) or deleting copies of the occurrences (*LessData*).

When an occurrence disappears (no occurrence copies in the system), the corresponding observer produces a new delay (using IF dense clocks). Then, this observer becomes available to monitor new occurrences. At the end the number of observers is bounded by the maximal numbers of occurrences present at the same time in the system. We only consider here finite systems where the number of occurrences is bounded.

Observers are implemented using IF observers. An IF observer is an IF process similar to a classic IF process with earmarks:

- synchronous reaction on events and conditions of observed system;
- observer process has the highest priority in the IF system;
- observer is deterministic;
- observer do not modify the behavior of the observed system (no output signal, no modification of system data).

7.3. Variation points modeling

To characterize the system, we need to explain all the parameters that impact the performance and the correctness of the system. Some are *a priori* fixed, due to technical or applicative requirements, some are just constrained and may be fixed. We first list all the parameters, then we show how to use them to drive contextual modeling and timing verification of the system.

```

intrusive observer occLifeCycle ;

    type T_Occ = array[maxOccNumber] of integer; /* counter of occurrences in the
system */
    type T_Counters = array[MaxDevNumber] of T_Occ; /* counters for each device
*/
    type T_clk = array[maxOccNumber] of clock; /* clocks for delay observation */
    type T_clks = array[MaxDevNumber] of T_clk; /* clocks for each devices */

    var data_obs SENSOR_DATA; /* data, occ number and ID sensor */
    var counters T_Counters;
    var clks T_clks;

state init #start ;
    ... /* initialization phase for counters */
    nextstate observe;
endstate;

state observe;
    match output newData(data_obs); /* new occurrence in the system */
    informal "initData : 1";
    task (counters[data_obs.idSensor-1])[data_obs.numOcc]:=1;
    set clks[data_obs.idSensor-1][data_obs.numOcc]:=0;
    nextstate -;
    match output moreData(data_obs); /* copy of occurrence in the system */
    informal "moreData : ++";
    task counters[data_obs.idSensor-
1][data_obs.numOcc]:=counters[data_obs.idSensor-1][data_obs.numOcc]+1;
    nextstate -;
    match output lessData(data_obs); /* deleting a copy of occurrence in the system */
    informal "lessData : --";
    task counters[data_obs.idSensor-
1][data_obs.numOcc]:=counters[data_obs.idSensor-1][data_obs.numOcc]-1;
    if counters[data_obs.idSensor-1][data_obs.numOcc]=0 then
/* no more copy in the system */
    task (({System}0).available_Occs[data_obs.idSensor-
1][data_obs.numOcc]:=false; /* free occ number */
    task counters[data_obs.idSensor-1][data_obs.numOcc]:=-1;
/* no copy for this occ number */
    reset clks[data_obs.idSensor-1][data_obs.numOcc];
/* no clock for this occ number */
    endif
    nextstate -;
endstate;
endobserver;

```

Table 7.3. *IF observer for occurrence in the system*

7.3.1. Variation points definition

This section describes the different variation points in the use and in the construction of a software data acquisition component. The variation points depend on the environment of the component and on their internal parameters.

The environment of the driver component is characterized by the actors interacting with it: the sensors (and its corresponding communication interface) and the RT applications. The first variation point in the environment is then defined by scalability factors:

- *NbSensors* represents the maximum number of sensors producing data for the driver;
- *NbApplicativeTasks* represents the maximum number of tasks reading data.

For each actor, it is necessary to describe its behaviors, from Driver component point of view. For CommunicationInterfaces/Driver interaction, the useful information is the arrival pattern of data and its associated delay, between data producing and the data getting by the software driver.

Considering the sensors, data production pattern is either periodic or sporadic or follows a burst law. These modeling views are classical for the real-time software community. For our demonstration, we consider simple cases where delay induced by communication interface is constant and there is no data lost by the communication interface.

To consider more complex characteristics, the work presented here should be extended to consider the impact of complex communication interface, but it out of the scope of this paper.

At the end, for a set of sensors $\{S_i\}$, the considered parameters are:

- $[m_i, M_i]$: represents the time interval for separation time between two occurrences in output of the communication interface associated with S_i ;
- D_i : represents the constant delay of data in output of the communication interface associated with S_i .

In the case of multiple sensors managed by the same communication interface (CI), the hypothesis is that two occurrences of two different items of data (produced by different sensors) are never produced at the same time, there is always a minimum delay between two occurrences, even for different sensors. Another parameter is then:

- dd_{CI} : represents the minimal separation time between two data produced by two different sensors S_i and S_j , in output of the communication interface CI.

For RTApplications/driver interactions, the useful information is the date of *read()* operations. Applications are classically modeled as a set of sporadic or periodic tasks $\{T_i\}$, performing *read()* operations after a certain amount of time. This time may be variable and is characterized by an interval. The classic real-time parameters are then defined by:

- $[B_i, E_i]$: represents the time interval for minimal and maximal separation time between two activations of task T_i ;
- $[readMini, readMaxi]$ represents the minimum and the maximum time between the instant *ApplicationReader* begins its execution and the instant *ApplicationReader* performs a read operation.

Following these modeling patterns, *periodic* MARTE *arrivalPattern* is a particular case of sporadic pattern where $m_i = M_i$, or $E_i = B_i$.

Once the actors are described, the internal parameters, characterizing different implementation of drivers must also be defined. From a generic point of view, these parameters describe the activation law of the driver's tasks and routines, and the chosen data storing policies. If the modeling approach is independent of the driver implementation, the exact list of internal parameters is not independent. These parameters are specific for each implementation.

To show the interest of the approach, we consider a specific (but classic) architecture for the driver's code, an example of which is presented in section 7.2.1. Then the given following parameters are a representative list, not an exhaustive and definitive list. They correspond to MARTE attributes that may be tuned at design stage:

- *arrivalPattern* of *ITMessage* handler: *aperiodic* or *burst* for activation by the sensor interrupt or *periodic* for an independent periodic activation by a specific periodic timer. If *arrivalPattern* is *periodic*, *timerPeriod* represents the period of the driver;
- [*itMin*, *itMax*] represents the minimal and the maximal execution time of *ITMessage* handler;
- [*opMin*, *opMax*] represents the minimum and the maximum time between the instant *DriverTask* reads a new message and the instant *DriverTask* stores it in *dataBuffer*;
- *waitingPolicyElements* policies for *SensorMessage*:
 - *messageQueueCapacityElements*: specifies the upper limit of message number allowed in a queue,
 - *InfiniteWaitWrite*: infinite waiting for write operation if the mailbox is full, otherwise no wait,
 - *InfiniteWaitRead*: infinite wait for read operation if the mailbox is empty, otherwise no wait;
- *waitingPolicyElements* policies for *dataBuffer*:

- consuming: application consume sensor information after reading it, or not,
- deleting: driver erase old sensor information before writing new one, or store, in a FIFO mode, the two lasts information.

Moreover, for each system entity owning a behavior, we add:

- *startTime*: explains the instant the entity starts in the system.

7.3.2. CDL implementation

This section describes how to integrate these different variation points using the CDL approach. The top level CDL diagram (see Figure 7.3a) defines 3 CDL2 nodes where we can describe variation points for *RTApplication*, one sensor and two sensors. Figure 7.3b describes *ApplicationConfiguration* CDL2 node. There is an alternative node which enables us to define 3 *RTApplication* periods.

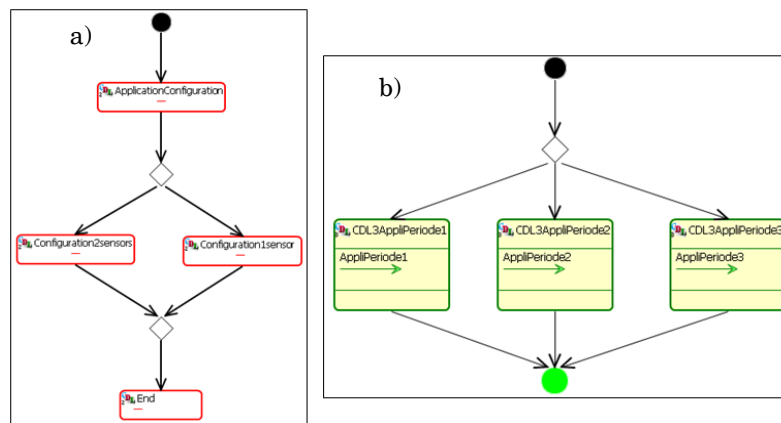


Figure 7.3. (a) Top level CDL diagram; (b) application configuration

Figure 7.4a describes *Configuration2Sensors* CDL2 node. There is an alternative node which enables us to define the time interval for separation time between two occurrences in output of the communication interface associated with each sensor.

Figure 7.4b describes *Configuration1Sensor* CDL2 node. There is an alternative node which enables us to define the time interval for separation time between two occurrences in output of the communication interface associated with one sensor.

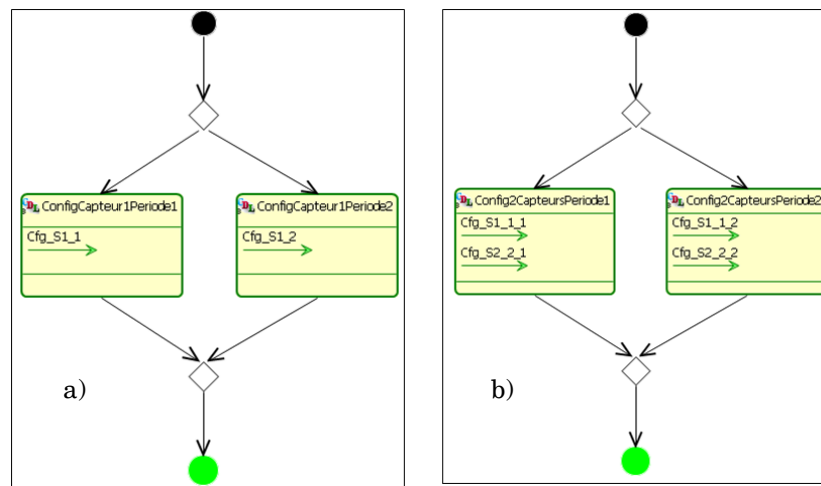


Figure 7.4. (a) *Configuration1Sensor*, (b) *configuration2Sensor*

Initialization system phase verification is out of the contextual verification stage. As usual in this domain, all initializations are made statically after the system starts. They are considered as corrects: there is enough memory and are well characterized. Then a main phase where the user reads data is started. The end phase is optional here, the system is considered to always be running.

In this chapter, initialization, made by the main function, called at the beginning, is a way to express all the internal parameters. Initialization is then a configuration phase.

After the configuration phase, all the modeled entities are ready to start. For active elements (sensors and tasks), starting time depends on the *startTime* configuration parameter.

7.4. Experiments and results

7.4.1. Tools

Using such an approach involves managing three multiple different views: IF modeling of the system, IF observers for timing evaluation and CDL context description. From an engineering point of view, managing several different views is complex and thus needs to propose automatic modeling and transformation tools.

For IF modeling of the system, we can use existing high-level UML-based tool like IFx [GRA 04] [IFx] which can produce IF files. The new MARTE OMG profile [OMG 09] is the most adapted modeling tool to describe the system: the *HRM package* defines all the necessary concepts for HW description, the *SRM package* defines all the necessary concepts for SW description, and the *NFP package* gives the elements for variation parameters.

However, for the moment, there is no automatic transformation tool from MARTE to IF. So, we are working on an *UMLMARTE2IF* tool, but at the present time, system modeling is performed directly using IF tools.

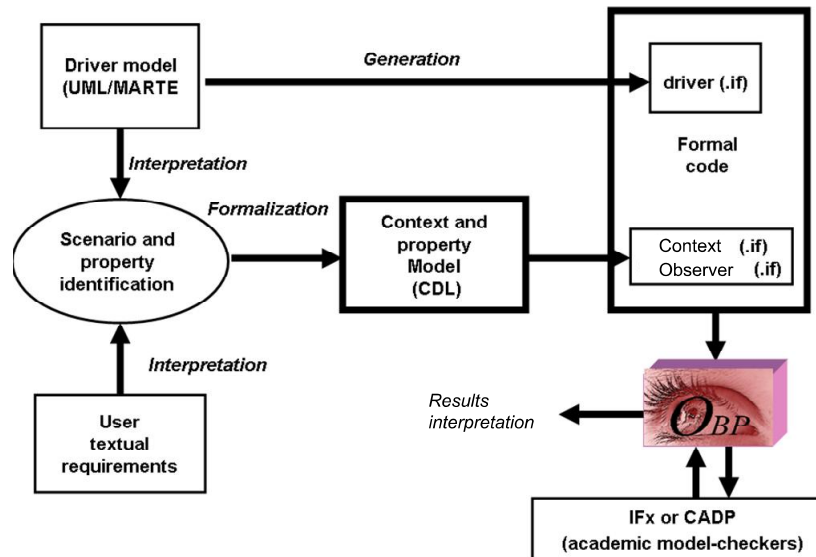


Figure 7.5. Methodology and toolset

To carry out our experiments, we implemented the OBP tool onto the Eclipse platform through plug-ins. OBP takes as input the CDL model and transforms it into IF2 automaton (see Figure 7.5). The essence of a translational approach to semantics is to move to a technological space that has a precise semantics [CLA 04] and tools. OBP leverages existing academic simulators and model checkers, as IFx. From CDL context diagrams, the OBP tool generates set context path automata which represent the set of the environment runs. OBP generates all the possible paths.

Each path represents one possible interaction between the model and context. The OBP tool generates, with a similar model transformation technique, the observer automata from the properties. Each generated context path is transformed into an IF2 automaton which is composed with the driver model under study and the generated observer automata by the IFx simulator.

To validate the component model, it is necessary to compose each path with the model and the observers. Each property must be verified for all paths. The accessibility analysis is carried out on the result of the composition between a path, a set of observers and the driver model. If there is a *reject* state reached of a property observer for one of the paths, then the property is considered as false.

At last, from Labeled Transition System and States files, produced after IF exhaustive simulation, we develop specific tools to automatically extract delays.

7.4.2. Experimentations

To show the interest of the proposed modeling approach, this section presents and discusses several results for the case study. In particular, we present the result of different partitioning steps.

Here we focus on a basic configuration with only one sensor. When data is lost, the *AbsenceOverflow* property is “false”. Table 7.4 shows that this property is “false” when *SENSOR_PERIOD* and *PERIOD_MAX_APPLI* is equal to 100 (Config; 1). The question is why the property is “false”? So, we must examine the LTS produced by IFx tool. There are several cases where the property is “false”. For instance when the driver clock is 100, the sensor can write a new value in the register before the driver reads it.

Below, we show how several IF simulations allow us to exhibit very different delay laws for the same driver architecture, but with different driver usages. Extraction delays are obtained using CADP [GAR 07]. We give now several contextual and implementation parameters, giving the following results. We consider a multi-periodic case with a unique sensor which periodically produces new information. The driver periodically polls the

CommunicationInterfaceRegister and the application also periodically reads data (provided by driver).

Parameter configuration	Config. 1	Config. 2
SENSOR_PERIOD	10	10
PERIOD_MIN_APPLI	6	8
PERIOD_MAX_APPLI	8	10
<i>AbsenceOverflow</i> property value	true	false
Number of transitions	393	522
Number of states	245	334

Table 7.4. *Test results*

Configuration	Sensor Period [mi,Mi]	Driver activation [itMin,itMax]	Application period [Bi,Ei]	[Minimal, Maximal] delay
C1	[20,20]	[20,20]	[130,130]	[3,11]
C2	[130,130]	[130,130]	[20,20]	[3,121]

Table 7.5. *Delay observation for two specific configurations of case-study*

NbSensors = 1, NbApplicativeTasks = 1;
Di = 0, readMini=readMaxi = 1, opMin=opMax = 1 and readMini = readMaxi = 1;
for each entity startTime = 0;
messageQueueCapacityElements = 3, InfiniteWaitWrite = false
(no wait) and InfiniteWaitRead = true;
Consuming = false (application does not delete read data) and
Deleting = true (saving last receive value)

We test the case where the driver and the application share the same period (130), respectively the sensor and the driver (130). Table 7.5 gives minimal and maximal delays for these two cases. Minimal delay is the same, corresponding to when sensor handler, driver and application tasks starts at the same time. Here maximal delay is related to sensor frequency.

For control theory, minimal and maximal values may not be sufficiently accurate to characterize timing behavior. So, we analyze the delay law (Figures 7.6 and 7.7) of such configurations, giving a sequence of possible delays through a state/transition diagram. The first state is state 0. Important transitions represent application reading data operation with its dynamic number of occurrences (given by observer) and the corresponding delay (occDelay).

In the first case, the delay simply alternates between 3 units of time (Sensor, Driver and Application start at the same time, the data is stored in the buffer after 2 units of time, Application takes 1 unit of time to read it) and 11 units of time (Application starts 10 units of time after Sensor and Driver and takes 1 unit of time to read the buffer) (only two possible delay values). In the second case, the delay law is more complex. The first reading operation is performed at instant 141 (first data is produced by Sensor at instant 130, arrives in the buffer at instant 132 and Application reads it at instant $(140 + 1)$ with a delay of 11). Then, Application reads the same data until new data is written in the buffer, so the delay cyclically increases by 20 modulo 130 (delay = $11 + k \cdot 20$ or $21 + k \cdot 20$ and delay is less than 121). A specific case occurs when the application starts at the same time as the sensor handler and driver (delay is 2 more due to scheduling effects).

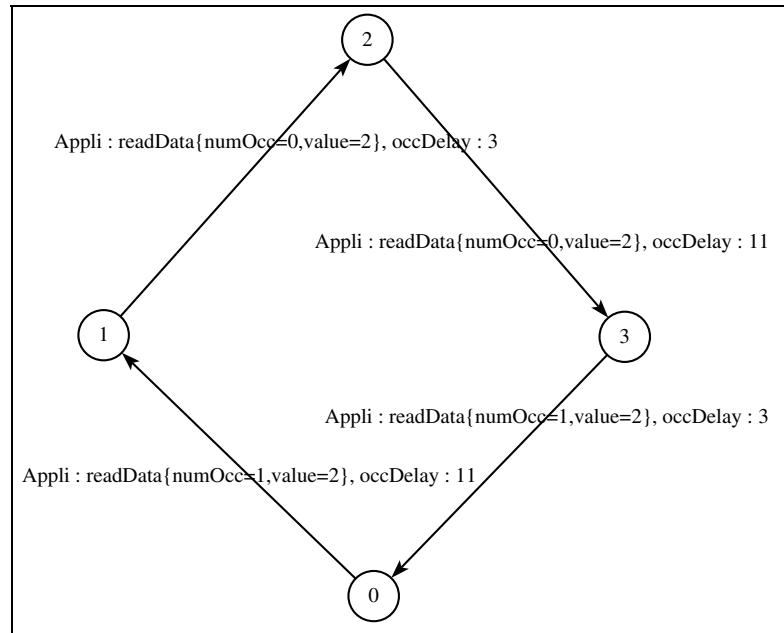


Figure 7.6. Delay law for configuration C1 (see Table 7.5)

7.5. Conclusion

The paper presents a Model-Driven approach based on heterogeneous modeling techniques (MARTE for design, CDL for contextual usage and parameterization, CDL for safety properties, IF for timing behavior, IF observers for delay evaluation) and tool (OBP for CDLtoIF generation, IFx for exhaustive simulation and CADP for behavior analysis) usage, in order to qualify, from a timing point of view, data acquisition drivers models. The proposed modeling approach serves both to qualify a data acquisition driver, that is formally described, and then to evaluate different implementations, for a certain context.

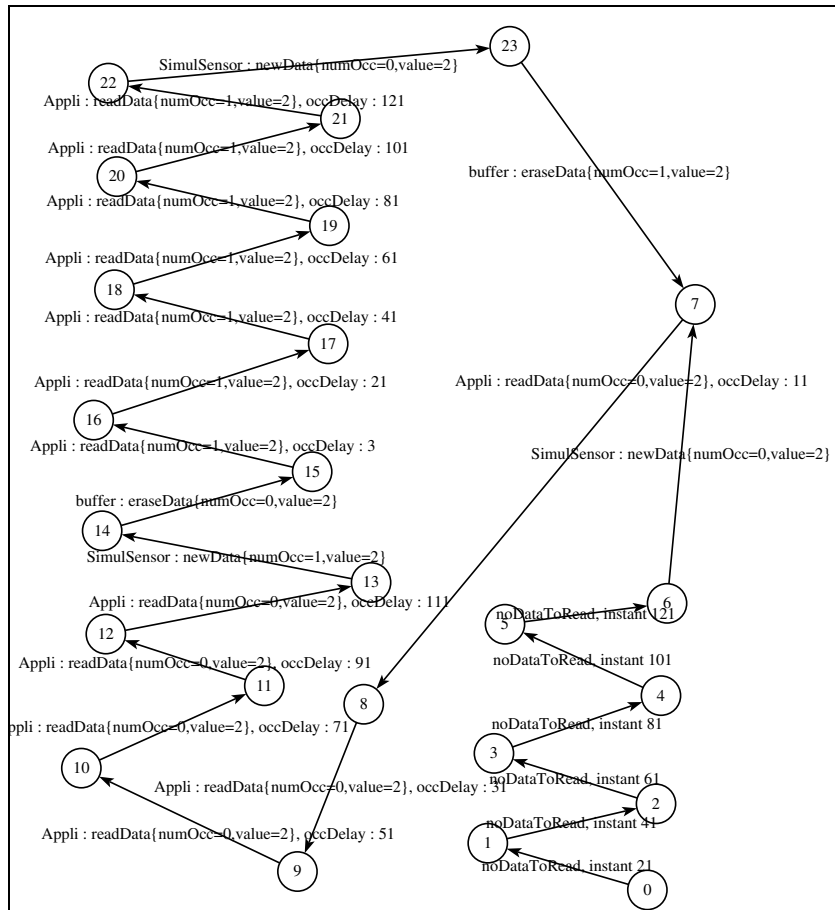


Figure 7.7. Delay law for configuration C2 (see Table 7.5)

7.6. Bibliography

- [AUD 93] AUDSLEY N.C., BURNS A., RICHARDSON M.F., TINDELL K., WELLINGS A.J., “Applying new scheduling theory to static priority pre-emptive scheduling”, *Software Eng. J.*, Vol. 8(5), pages 284–292, September 1993.
- [ALU 94] ALUR R., DILL D.L., “A theory of timed automata”, *Theoretical Computer Science*, vol. 126, pp. 183-235, 1994.

- [BEL 04] BELARBI M., BABAU J.P., SCHWARZ J.J., “Temporal verification of real-time multitasking application properties based on communicating timed automata”, in *DS-RT '04: Proceedings of the Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04)*, pages 188-195, Washington, DC, USA: IEEE Computer Society, 2004.
- [BER 03] BERNAT G., “Response time analysis of asynchronous real-time systems”, *Journal of Real-Time Systems*, 25 (2-3), pages 131 – 156, 2003.
- [BOZ 02] BOZGA M., GRAF S., MOUNIER L. “IF2: A validation environment for component-based real-time systems”, in *Proceedings of Conference on Computer Aided Verification, CAV'02*, Copenhagen, LNCS. Springer Verlag, 2002.
- [BEN 05] Ben HÉDIA B., JUMEL F., BABAU J.P., *Formal Evaluation of Quality of Service for Data Acquisition Systems, FDL'05*, pages 579-588, Lausanne, September 2005.
- [BAT 03] BATE I., NIGHTINGALE P., CERVIN A. “Establishing timing requirements for control loops in real-time systems”, in *Microprocessors and Microsystems*, Vol 27(4), pages 159-169, 20 May 2003.
- [BOR 97] BORNOT S., SIFAKIS J., TRIPAKIS S., “Modeling urgency in timed systems”, in *International Symposium: Compositionality – The Significant Difference*, Malente (Holstein, Germany), September 1997. Lecture Notes in Computer Science 1536, Springer Verlag.
- [CLA 04] CLARKE T., EVANS A., SAMMUT P., WILLIAMS J., *Applied Meamodeling: A foundation for Language Driven Development*, Technical report, version 0.1, Xactium, 2004.
- [DHA 08] DHAUSSY P., AUVRAY J., DE BELLOY S., BONIOL F., LANDEL E., “Using context descriptions and property definition patterns for software formal verification”, *Workshop Modevva'08*, 9 April 2008 (hosted by ICST 2008), Lillehammer, Norway.
- [DHA 09a] DHAUSSY P., CREFF S., PILLAIN P.Y., LEILDE V., *CDL language specification (Context Description Language)*, technical report version N° DTN/2009/8, Ensietia, 5 March 2009.

- [DHA 09b] DHAUSSY P., PILLAIN P.Y., CREFF S., RAJI A., LE TRAON Y., BAUDRY B. “Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation”, in *Lecture Notes in Computer Science 5795*, Springer Verlag, Andy Schuerr, Bran Selic (Eds): *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models’09)*, No 5795 , pages 438-452, 2009.
- [DWY 99] DWYER M.B., AVRUNIN G.S., CORBETT J.C. “Patterns in property specifications for finite-state verification”, in *Proc. of the 21st Int. Conf. on Software Engineering*, pages 411-420. IEEE Computer Society Press, 1999.
- [FOK 02] FOKKINK W., IOUSTINOVA N., KESSELER E., VAN DE POL J., USENKO Y. YUSHTAIN Y.A., “Refinement and verification applied to an in-flight data acquisition unit”, in *13th Conference on Concurrency Theory CONCUR’02*, Brno, Czech Republic, Lecture Notes in Computer Science 2421, Springer, pages 1-23, 2002.
- [GAR 07] GARAVEL H., MATEESCU R., LANG F., SERWE W., “Cadp 2006: A toolbox for the construction and analysis of distributed processes,” in *CAV, ser. Lecture Notes in Computer Science*, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, pages 158–163, 2007.
- [GRA 04] GRAF S., OBER I., “Model-checking UML models via a mapping to communicating extended timed automata”, in *Proceedings of SPIN’04*, Barcelona, Spain, April 2004.
- [HAU 05] HAUGEN O., HUSA K.E., RUNDE R.K., STOLEN K., “Stairs: towards formal design with sequence diagrams”, in *Journal of Software and System Modeling*, 2005.
- [HIE 03] HIEU P.T., GÉRARD S., LUGATO D., TERRIER F., “Scheduling validation for UML-modeled real-time systems”, *Euromicro Conference on Real-Time Systems*, WIP session, Porto, Portugal, 2003.
- [HAL 93] HALBWACHS N., LAGNIER F., RAYMOND P., “Synchronous observers and the verification of reactive systems”, in *3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST’93)*, 1993.
- [IFx] IFx tool description, OMEGA project, <http://www-if.imag.fr/>

- [JAN 99] JANSSEN W., MATEESCU R., MAUW S., FENNEMA P., STAPPEN P., "Model checking for managers", *Conference Spain'99*, p. 92-107, 1999.
- [KON 05] KONRAD S., CHENG B., "Real-time specification patterns", in *Proc. of the 27th Int. Conf. on Software Engineering (ICSE05)*, St Louis, MO, USA, 2005.
- [KLE 93] KLEIN M. & all, *A Practioner's Handbook for Real-Time Analysis*, Kluwer Academic Publishers, 1993.
- [FEN 01] FENG-LI L., Analysis, Design, Modeling, and Control of Networked Control Systems, PhD thesis, University of Michigan, 2001.
- [LIU 73] LIU C.L., LAYLAND J.W., "Scheduling algorithms for multiprogramming in a hard real time environment", *Association for Computing Machinery*, Vol. 20(1), pages 46-61, 1973.
- [LIM 04] LIME D., ROUX O., "A translation based method for the timed analysis of scheduling extended time Petri nets", *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, 187-196, Lisbon, Portugal, 2004.
- [MAN 92] MANNA Z., PNUELI A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer, New York, 1992.
- [MER 00] MERILLON F., REVEILLERE L., CONSEL C., MARLET R., MULLER G., "Devil: an idl for hardware programming", *4th USENIX OSDI Symposium*, pages 17-30, 2000, San Diego, California, USA, October 23-25.
- [MIG 02] MIGGE J., JEAN-MARIE A., NAVET N., "Timing analysis of compound scheduling policies: Application to posix1003.b", *Journal of Scheduling*, Kluwer Academic Publishers, 6 (5), pages 457-482, 2002.
- [MOR 09] MOREL L., BABAU J.P., BEN-HEDIA B., "Formal modelling framework of data acquisition modules using a synchronous approach for timing analysis", in *Proceedings of the 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software WRTP-RTS'09*, pages 123-130, Mrongovia, Poland, October 2009.

- [OMG 09] Object Management Group, A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems), Version 1.0, OMG doc. formal/2009-11-02, December 2009.
- [OSEK] OSEK/VDX-operating systems for automotive applications, version 2.2.3, February 2005, <http://portal.osek-idx.org/files/pdf/specs/os223.pdf>
- [PET 99] PETTERSSON P., Modeling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice, PhD Thesis, Uppsala University, February 1999.
- [POSIX] ISO/IEC Standard 9945-2003 [IEEE Std 1003.1, 2004 Edition] Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language].
- [RAM 99] RAMANATHAN P., “Overload Management in Real-Time Control Applications Using (m, k)-Firm Guarantee”, *IEEE Transactions on Parallel and Distributed Systems*, Vol 10(6), pages 549-559, 1999.
- [SAN 05] SANDBERSON M., TÖRNGREN M., WIKANDER J., “The Effect of Randomly Time-Varying Sampling and Computational Delay”, *Proceedings of the 16th IFAC World Congress*, Vol. 16(1), 2005.
- [SCH 97] SCHEIDLER C., HEINER G., SASSE R., FUCHS E., KOPETZ H., TEMPLE C., “Time-Triggered Architecture (TTA)” presented at *EMMSEC'97*, Florence, Italy, Nov. 1997, published in *Advances in Information Technologies: The Business Challenge*, IOS Press.
- [SIF 96] SIFAKIS J., YOVINE S., “Compositional specification of timed systems”, in *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, pages 347-359, Grenoble, France, February 1996. Lecture Notes in Computer Science 1046, Springer-Verlag.
- [TIN 94] TINDELL K., CLARK J., Holistic, “Schedulability analysis for distributed hard real-time systems”, *Euromicro Journal*, Vol 40, pages 117-134, 1994.
- [TOR 98] TÖRNGREN M., “Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems”, *Real-Time Systems*, Vol. 14(3), Pages 219 – 250, May 1998.

- [WAS 08] WASZNIOWSKI L., HANZ'ALEK Z., "Formal verification of multitasking applications based on timed automata model", *Real-Time Systems*, Vol. 38(1), pages 39–65, 2008.
- [WHI 05] WHITTLE J., "Specifying precise use cases with use case charts", in *MoDELS'06*, Satellite Events, pages 290–301, 2005.
- [WIT 95] WITTENMARK B., NILSSON J., TORNGREN M., *Timing Problems in Real-time Control Systems*, 1995.

Chapter 8

SoC/SoPC Development using MDD and MARTE Profile

8.1. Introduction

Thanks to the ever-increasing performance of digital electronics, an entire embedded system can now be integrated on a single chip: i.e. a SoC – *System on Chip* – or a SoPC – *System on Programmable Components* – for FPGA – *Field Programmable Gate Array* – reconfigurable components.

In parallel, to catch up with this components complexity, a dramatic enhancement of hardware design productivity is required to avoid a “productivity gap” [ITR 07]. ESL – *Electronic System Level* – tools have emerged in order to tackle this issue by improving the level of abstraction of hardware developments. For example, some ESL tooling, enable us to simulate a design at TLM – *Transaction Level Modeling* – with SystemC language or to synthesize hardware architecture

Chapter written by Denis AULAGNIER, Ali KOUDRI, Stéphane LECOMTE, Philippe SOULARD, Joël CHAMPEAU, Jorgiano VIDAL, Gilles PERROUIN and Pierre LERAY.

directly from C functional code rather than from a RTL – *Register Transfer Level* – description.

In addition to ESL modeling approaches, UML – *Unified Modeling Language* – [OMG 06b] originally dedicated to Software development has extended its scope to System or real time embedded application development through SysML – *System Modeling Language* – [OMG 08a] and MARTE – *Modeling and Analysis of Real-Time Embedded systems* – [OMG 07] profiles.

Moreover, MDA – *Model Driven Architecture* – [OMG 03], promotes a development methodology based on model transformations at several levels of abstraction and that follows the well known Y-Chart co-design approach: at each level, a PIM – *Platform Independent Model* – representing the application is mapped onto a PM – *Platform Model* – representing the target architecture to obtain a PSM – *Platform Specific Model* – representing the implementation.

As the development of SoC/SoPC components covers system, software and hardware engineering activities, from the system requirement capture, up to the fine analysis of the hardware logic timing, a SoC/SoPC development methodology should take advantage of these new UML profiles and MDA methodology in capitalizing and the achievements of the ESL community.

An experimentation in this approach has been carried out in the frame of the MOPCOM SoC/SoPC project [MOP 07, KOU 08] and is presented in this chapter. Section 8.2 is a state-of-art overview. The following sections present the different types of models identified in MOPCOM SoC/SoPC development methodology and the MARTE profile elements used at each level. An example of Cognitive Radio application is used to illustrate this process in section 8.4. Finally, sections 8.9 and 8.10 describe MOPCOM tooling developed to support

the design process and the generation of – *Hardware Design Language* (HDL) – as well as the automatic generation of documentation.

8.2. Related works

As explained above, developments of SoC/SoPC and RTES – *Real Time Embedded Systems* – in general is related to the ESL community. Only a reliable methodology, based on appropriate languages and tools can help to handle market pressure (time-to-market, competitiveness), increasing evolution of the technology or standards (DO-178B, DO-254, etc.) [GER 03] related to such developments.

In order to address the market constraints and obsolescence issues, separation of concerns is needed to allow the concurrent development of applications and execution platforms. This kind of approach was first proposed in the Gajski and Kuhn Y-Chart model [GAJ 83], generalized by the Model Driven Development approach and standardized by the Model Driven Architecture OMG's standard. Moreover, in order to allow faster design of space exploration, systems under study must be modeled and validated at several levels of abstraction [SAN 04]. There is no real consensus about the number of required abstraction levels even if there are some efforts to define and standardize abstraction levels and their related services and interfaces [KAS 07].

Several languages enable the description of behavioral or structural parts and the allocation of the system under development. The most important factors influencing the choice of a language in a modeling or analysis activity are its expressiveness power and its tooling. For instance, SystemC [OSC 05] is a language allowing functional and platform modeling at several levels of abstraction, and is supported by several free or commercial tools dedicated to analysis or compilation / synthesis.

In addition to the separation of concerns and definition of levels of abstraction, there is a need to favor reusability in order to improve the productivity. Indeed, large system development has to rely on libraries of proved and well-documented IPs at each level of abstraction. Interconnection and exchange between IPs is based on the use of standard interfaces and protocols. In some cases, Ad-doc IPs can be wrapped to conform to standards [KOU 05].

Developments of RTES include modeling activities, using languages based on either grammars or metamodels, as well as analysis activities such as formal validation or simulation. The main issues when modeling RTES are the description of concurrency / communication [GER 02], execution platform, possibly represented at several levels of abstraction, and QoS – *Quality of Service*. Modeling and analysis activities must be replaced in the context of a well-defined methodology. For that, there are two different approaches:

- use several DSL – *Domain Specific Language* – fitting for each modeling or analysis activity,
- use a general purpose modeling language, such as UML, with dedicated profiles to support the required concepts.

Additional mechanisms such as annotations are also required in order to add relevant information needed by analysis tools (example: resource usage for schedulability analysis).

Based on the use of selected formalisms, several methodologies and tools have been developed to support RTES development. Few examples are given below.

The methodology MCSE – *Méthodologie de Conception des Systèmes Electroniques* – [CAL 08] proposed by the University of Nantes, enables design space exploration through the use of the SystemC TLM language. The French company Cofluent Design [COF], in partnership with MCSE Team, has

developed the ESL design tool “Cofluent Studio”, based on Eclipse environment, which supports MCSE methodology.

The Ptolemy environment from UC Berkeley [BUC 02] allows description of systems mixing several MoC – *Models of Computation* – through the notions of actors and directors. A director defines a domain of execution for its actors enabling the mixing of several models of computation in the same model. This is an important issue because real-time systems usually mix analog and digital devices and possibly several time domains.

Syndex from INRIA is a tool implementing the Architecture Adequation Algorithm [KAO 04] addressing the allocation issue.

In the context of the UML, several profiles have been proposed to extend UML capabilities in order to handle modeling and analysis of RTES or SoC. Among them, we can cite:

- the SPT – *Schedulability Performance Time* – OMG’s profile [OMG 05a] which was based on the version 1.4 of UML and completed by the QoS and Fault Tolerance OMG’s profile [OMG 08b] for the non-functional aspects,
- the UML4SOC OMG’s profile is dedicated to describe SoC [OMG 05b],
- the UML for SystemC profile proposed in [RIC 05] gathers the capabilities of UML and SystemC,
- the UML for MARTE OMG’s profile [OMG 07] can be viewed as an improvement of the SPT profile (see section 8.3),
- the Gaspard profile is specific to parallel and distributed computing applications implemented into SoC [PIE 08].

Based on the use of UML profiles, examples of RTES or SoC design environments are given below.

The *ACCORD/UML* methodology [GER 02] aims at using UML concepts to design RTES. It was originally based on the use of the SPT profile and now relies on MARTE profile. It is supported by the eclipse-based PAPYRUS tool [CEA].

The University of Milan, in collaboration with STMicroelectronics, proposes a development process for embedded systems and SoC called UPES – *Unified Process for Embedded Systems* –, based on UML for SystemC profile [RIC 07].

The Gaspard Methodology [PIE 08] is intended to provide a framework for developing parallel and distributed applications implemented on SoC. This methodology is an implementation of the MDA approach in the eclipse framework and provides a set of transformation rules allowing generation of optimized SystemC Code for repetitive structure architecture.

8.3. MOPCOM process and models

Our SoC/SoPC development process is based on a MDD approach. It relies on two elements: a conventional System Engineering methodology for the System Analysis, including Requirement Capture and Functional Analysis activities, and a dedicated process for the SoC/SoPC architecture definition. This process emphasizes:

- separation of concerns: as explained in the previous sections, separation of the application (PIM) from the platform (PM) and description of the mapping to generate the implementation (PSM), as in a classic Y-chart co-design pattern;
- analysis Driven Development: the previous pattern can be reproduced with several execution platform abstractions, depending on the kind and the number of analysis we need to perform. For instance, in the MOPCOM methodology, we

have define three abstraction levels for the target platform as explained below.

The set of input/output models of our process is presented in Figure 8.1.

– AML – *Abstract Modeling Level* – is intended to describe a virtual execution platform where expected concurrency, pipeline and communication policy are explicitly defined. The allocation model describes the mapping of the application onto the Model of Computation defined by the AML platform.

– EML – *Execution Modeling Level* – provides a coarse definition of the physical platform. At this stage, the platform is composed by generic components such as Processing or Memory Units. Allocation model describes the mapping between the previously generated model onto the platform.

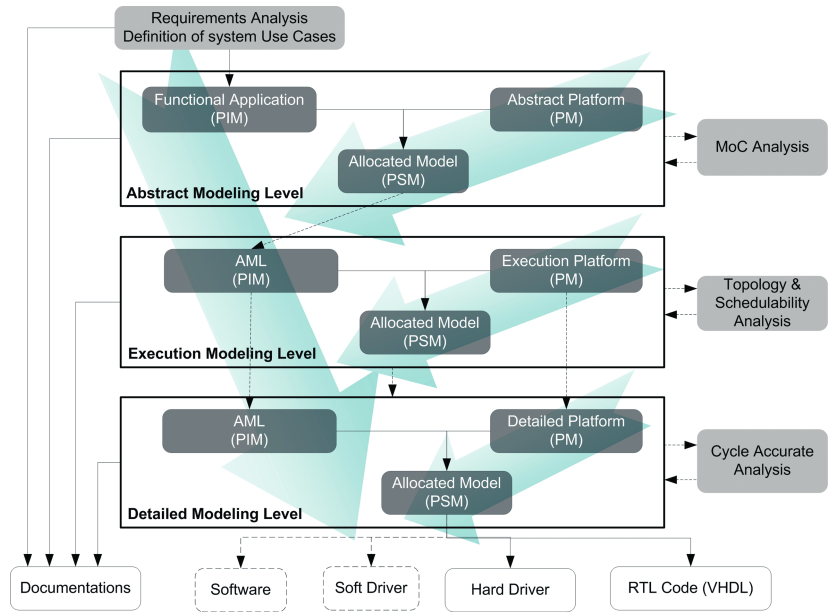


Figure 8.1. MOPCOM abstraction levels and included models

This results in a topology description allowing performance or schedulability analysis.

– DML – *Detailed Modeling Level* – refines the definition of the platform with the necessary information for Software and RTL HDL code generation. The Allocation consists of mapping the previous PSM AML model onto a more precise description of the platform.

In addition to automatic code generation, other artefacts, such as documentation, can be produced automatically.

Each stage is validated against simulations. The main benefits of such an approach are the minimization of the risks as emphasized in a classic iterative development process [BOE 88] and the optimization of the development time, through concurrent developments.

Another goal of our work is to evaluate the relevance of the UML MARTE profile in terms of concepts: the MARTE profile encloses a large set of concepts to model and analyze Real Time Embedded Systems. These concepts are organized into several hierarchical sub-profile packages, following concerns they are related to.

For instance, the design of RTES deals with the modeling of properties that quantify offered and provided services by a resource to the application. For this purpose, there is a package dedicated to the description of NFP – *Non Functional Properties*. The NFP constraints are applied all along the process from the capture requirement to the detailed platform modeling. Properties and constraints can be expressed using the VSL – *Value Specification Language* – which can be viewed as an extension of OCL – *Object Constraint Language* – [OMG 06a] taking into account QoS characteristics.

Time models (causal, logical or continuous) and time access (logical or chronometric clocks) are also important issues

when modeling RTES. Concepts related to time have been introduced into the Time Package and refine the UML time model. The clocked or synchronous time abstraction divides the time scale in a discrete succession of instants while the physical time can be viewed as dense discrete time, which is useful to model analogical devices through ODE – *Ordinary Differential Equation*.

Analysis activities deal mainly with qualitative or quantitative features of the system, such as period, occurrence kind, duration, jitter, etc. They are part of the HLAM – *High-Level Application Modeling* – package.

Analysis of schedulability or performance of a system is allowed in MARTE using the SAM – *Schedulability Analysis Modeling* – and PAM – *Performance Analysis Modeling* – packages through the concepts of workload, resource acquisition and release and so on. By default MARTE allows two kinds of analysis: performance and schedulability but it provides generic concepts in order to allow us to define other kinds of analysis, such as WCET – *Worst Case Execution Time*.

Platforms can be described at several levels of abstraction with more or less details, depending on the kind and the number of analyses to be performed (see above). The GRM package – *General Resource Modeling* – provides concepts allowing the modeling of the platform at a high abstraction level while those concepts are refined in the SRM and HRM packages. The SRM package – *Software Resource Modeling* – provides details related to the description of software platforms such as RTOS or middleware. The HRM package – *hardware resource modeling* – provides concepts related to the description of physical platform such as Buses, FPGA, Processor, etc.

The MOPCOM methodology provides the rationale to use all those MARTE concepts in a consistent manner. Indeed

to enforce the definition of our methodology, we select some MARTE elements for each abstraction level in our SoC/SoPC development process. We have defined the usage scope of those concepts by adding constraints to the metatype of the UML/MARTE language. Those set of constraints specialize the language (UML/MARTE) to the SoC/SoPC domain and are capitalized into the model of the process itself.

In the next sections, we present each level of our MDD process with its associated set of MARTE concepts.

8.4. Application

The development process presented above has been experimented through a CRS – *Cognitive Radio System*. A CRS is a piece of radiocommunication equipment which is able to adapt its functionality according to the Radio environment [MIT 01, HAC 07]. A CRS for instance can identify the RAT – *Radio Access Technologies* – available and determines their characteristics such as bandwidth and load as well as environment characteristics such as localization. The more suitable available RAT in the area is then selected and the CRS receiver chain is configured to communicate through the corresponding protocol.

This type of system is quite complex and for the demonstrator only two Use Cases are analyzed:

- the “Locate RAT Source” Use Case developed by Thales with Supelec localizes the Direction Of Arrival of the available RAT. The RAT identification is carried out after a Spectrum analysis of the RF environment and a blind standard recognition of the communication protocol. The detected radiocommunication signals are recognized by analyzing some discriminating parameters that characterize the protocol such as: channel bandwidth, frequency hopping, single/multicarrier signal, etc. The localization is carried out with a beamformer

and removes the eventual multipaths to only keep the Direct Line-Of-Sight direction.

– the “Wireless transmission” Use Case developed by Thomson Corporate Research [LEB 08] implements a wireless stack based on standard 802.16 and uses the TDMA – *Time Division Multiple Access* – MIMO/OFDM – *Orthogonal Frequency Division Multiplexing* – technologies. For the demonstrator, the Use Case focuses on baseband processing and more specifically on the algorithms and the architectures of MIMO – *Multiple Input Multiple Output* – decoding.

The Cognitive Radio System is made up of 4 antennas, a baseband processing, and an Ethernet connection. The targeted platform for the implementation is a reconfigurable component that can demonstrate self-reconfiguration.

8.5. System analysis

The first activity of our design process is the System Analysis. This activity is not specific to SoC/SoPC development and can rely on an existing System Engineering methodology such as the Telelogic Harmony/SETM [ART 08] methodology. Harmony/SETM is a SysML based methodology that consists mainly of two steps: Requirement Capture and Functional Analysis to which we add MARTE improvements, essentially related to real-time features.

8.5.1. Requirement analysis

A Requirement Analysis captures system functional and non-functional requirements and merges them into Use-Cases. Use-cases define services provided by the system to external entities (actors).

The operational contracts (scenarios) and the interfaces between the actors and the system are formalized at high

abstraction level using for example textual descriptions or models such as Sequence Diagrams or Activity Diagrams.

At this stage, we use the stereotypes of the packages NFP, VSL, Time of MARTE. Figure 8.2 is an example of a Sequence Diagram related the Use Case “Locate RAT Source” that relies on «TimedInstantObservation» and «TimedConstraint» stereotypes. The TimedConstraint, specifying a duration constraint, is expressed using the VSL syntax.

8.5.2. Functional analysis

Compared to the requirements Analysis, the functional analysis focuses on the functional decomposition of the Use Cases. Use Cases are split into functions. This functional decomposition is captured through Class and Object Diagrams whereas the behavior of the Use Cases is defined with Activity, Statechart and Sequence Diagrams refining the internal interactions between the different functions.

At this stage, we use the same MARTE elements that in the previous one in order to precise for example derived

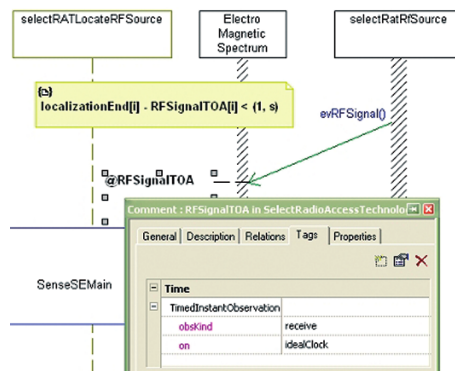


Figure 8.2. Sequence Diagram of UC “LocateRAT Source”

constraints and internal data types. Nevertheless specific Functional Analysis rules have been applied to ease allocation onto the platform in the next steps of the process through orthogonalization of specification and implementation and the use of some Design Patterns [GAM 95]:

- “Facade”: to unify configuration interfaces;
- “Decorator”: to separate concerns simulation and system behavior;
- “Singleton”: to share single object;
- “Strategy”: to model dynamic reconfiguration.

Figure 8.3 shows an example of the functional decomposition for the UC “Locate RF Source”.

8.5.3. Action language

To perform validation, Requirement Analysis and Functional Analysis, models must be executable. This requires an action language for low-level expressions that complements the high-level UML semantic and diagrams, and to specify operation bodies, trigger/guard/action on transition

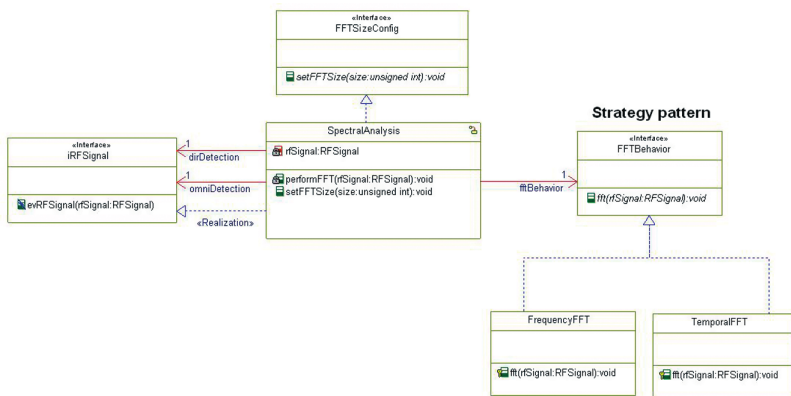


Figure 8.3. Class Diagram related to UC “Locate RF Source”

and in states, and data declarations. The selection of the most suitable action language raises questions about textual or graphical notation, and general versus HDL-specific language that should be accessible to system, software and hardware designers without too many problems related to its learning curve.

After analysis, C++ language turned out to be the most convenient choice. Indeed, it is a wide-spread and standard object-oriented language, supported by many high-performance development environments (including Rhapsody-in-C++ suite).

Only a C++ subset is used in the models (along with some macros for event and port handling). C language is fully mastered by hardware designers (similar syntax for non-OO subset) and next-generation SystemC language is basically a C++ library dedicated to hardware applications.

8.6. Abstract modeling level

While the aim of the functional analysis was the definition of the behavior of the system and its functional breakdown into functional blocks, the aim of this level is to identify the needed level of concurrency and define the way concurrent blocks communicate. The underlying goal of those identifications is analysis that can be performed like consistency or deadlocks analysis.

The notion of concurrency in UML is supported by the “isActive” meta-attribute meaning that corresponding classes manage their own thread of execution, but it does not say so much about what kind of interactions can occur between active classes. Still, there is missing information needed to make relevant analysis at this level. Fortunately, the MARTE profile completes the definition introducing the concepts of RTUnits or RTConnectors in the HLAM package.

The RTUnit is a refinement of the Block concept introduced in SysML with additional real-time features. An RTUnit provides and requires a set of real-time services. In order to realize those services, a RTUnit owns a set of real-time behaviors with bounded or unbounded message queue for each of those behaviors. An RTUnit can also own a set of schedulable resources which are typed by other RTUnits, connected through RTeConnectors, allowing hierarchical description of the system. Then, the owning RTUnit provides an execution context (domain) for each of these sub-RTUnits and is responsible for managing their interactions and concurrency.

Since the AML model provides an execution framework for the system under study, it can be considered as the highest abstraction of the execution platform. Indeed, its identification constitutes the first step of the design space exploration.

At this level, every concurrent unit is a stereotyped «RTUnit» characterized by its provided/required services set and its real-time behaviors. RTUnits communicate through real-time connectors «RTeConnector». Actions performed by an RTUnit are stereotyped «RTAction». For each of those concepts, quantitative or qualitative aspect information is provided: duration, priority, occurrence kind, etc.

The AML platform aggregates the set of functional objects needed to implement the system with additional information about concurrency and communication. To generate the real-time units, we need to provide the functional design and allocation with associated constraints. Figure 8.4 describes the process of allocation while Figure 8.5 gives, in MQL language (see section 8.9), an excerpt of the transformation rules that have been applied to generate the allocated platform.

Functional blocks are turned into AML blocks stereotyped «RTUnit» and «Allocated» for the purpose of traceability.

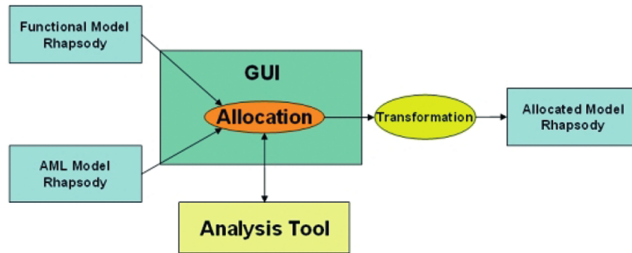


Figure 8.4. *Process of Allocation*

```

//references to the needed stereotypes in the target model
var clSt : rhapsody.Stereotype = target.getInstances("Stereotype").detect("name", "RTUnit");
var opSt : rhapsody.Stereotype = target.getInstances("Stereotype").detect("name", "RTService");
//Iterate over instances of the collaboration
foreach (i : mopcom.Instance in mcPack.getInstances("Instance")) {
  //detect corresponding RTUnit definition in the target model
  cl = target.getInstances("Class").detect("name", i.type.name);
  //if it does not exist, create it and add it in the target model
  if (cl == null)
  {
    rtu = i.type;
    cl = target.create("Class");
    cl.name = rtu.name;
    rtu#class.add(cl);
    //add "RTUnit" Stereotype to the generated class
    cl.stereotypes.add(clSt);
    cl.stereotype = clSt;
    //add the "RTUnit" in its owning package
    cl.owner = rhPack;
    rhPack.classes.add(cl);
  }
}

```

Figure 8.5. *Excerpt of Allocation to Allocated Model Transformation in MQL Syntax*

Connectors binding ports implement communications related to the MoC indicated on the allocation constraints of the link. Special blocks are inferred for (de)multiplexing data between RTUnits when several functional blocks are executed sequentially.

8.7. Execution modeling level

The MOPCOM Execution Modeling Level (EML) is made up of three different models (Figure 8.1). The main goal of this

level is to model the topology of the virtual hardware platform and to analyze the system scheduling.

8.7.1. The platform independent model/application model in EML

The PIM model in EML is similar to the PSM model in AML with a refactoring if necessary. We can transform the functional structure in order to allocate the PIM onto PM. In fact, if the analysis results do not respect the specifications, the functional structure or/and the topology of the virtual hardware platform must be changed.

8.7.2. The platform model in EML

In EML, PM only represents the topology of the virtual hardware platform based on high-level generic components. Indeed, the objective of the virtual platform is to hide the physical platform to the application. This PM cuts-out superfluous details such as the protocol description, the type of computing resources and storage resources used in the physical platform model. The initial interest of such a modeling is to represent the nodes of computation, of storage, of communication and the services offered by the platform to the application. A natural modeling concept of PM in EML is a transactional-level modeling, as promoted by Gajski and SystemC community in general. Thus the communications between the components of the platform are represented by calls of functions and not by a detailed model of the protocol and the connectivity which are represented in the RTL level. The MOPCOM methodological tool at this level is MARTE GRM – *Generic Resource Modeling* – sub-profile. Figure 8.6 shows an example of PM with the following stereotypes of MARTE: «ComputingResource», «StorageResource» and «CommunicationMedia».

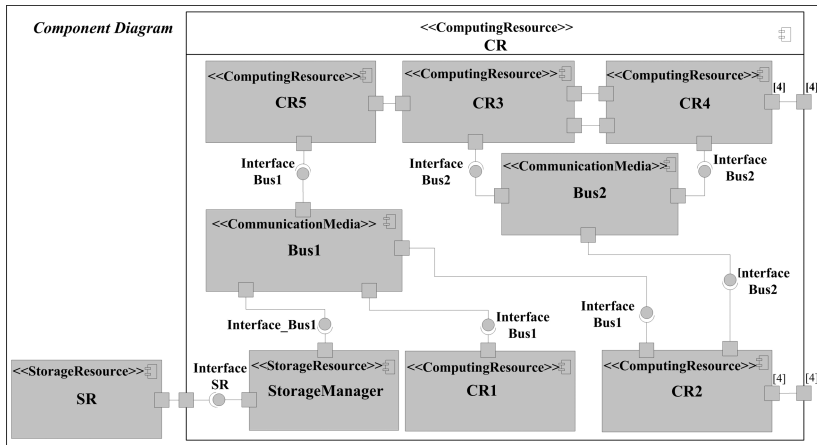


Figure 8.6. An example of PM in EML with MARTE stereotypes

8.7.3. The platform specific model/allocation model in EML

In the PSM, the MoC components (of the PIM) are mapped onto the components of the PM. The allocated methodology is the same as for AML (Figure 8.4). Moreover, the mapping of PIM onto the PM to form the PSM must not damage the semantics of the MoC. Actually, if more than two MoC components are mapped onto one component of the PM, the semantics of the point-to-point communication between the MoC components is not affected. But if two MoC components, which communicate between them, are mapped onto two different components of the PM, the semantic is not assured because the link of the communication between both hardware components can be a bus and not a point-to-point link. Therefore the semantics have to be guaranteed in a bus communication.

8.7.4. Analysis model

To analyze the scheduling and the performances of the system, some information must be added to the models of this level. What is the significance of schedulability analysis? It provides the ability to evaluate time constraints and guarantee worst-case behavior of a real-time system. For the schedulability analysis, MARTE SAM sub-profile is recommended. This sub-profile offers the elements to add annotations in the different views of the model in order to evaluate the scheduling. The way to use the SAM sub-profile is explained below:

- In the PIM, the behavioral descriptions are annotated by time constraints and by the size of exchanged messages with the stereotypes of SAM; the sequence diagrams is well adapted to add these annotations; the different scenarios of behavioral description form the workload behavioral model;

- In the PSM, the objects diagram is annotated in order to indicate the type of scheduling resource of each element of the model (Figure 8.7);

- The last step is to add a view in order to model the analysis context and the parametric analysis context. The analysis context model indicates the start and the stop conditions of the different behavioral scenarios. And the

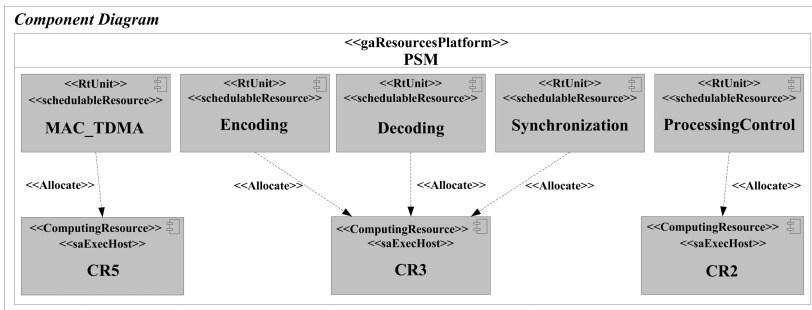


Figure 8.7. An example of PSM with SAM stereotypes of MARTE

parametric analysis context is an instance of this analysis context model in which the values are set in order to simulate the scheduling.

After these three steps, the SAM of the system is defined for EML. The MOPCOM process does not specify the process and the tool, such as the Cheddar [SIN 04] tool, used to analyze the scheduling. It should be noted that the metamodels of UML and MARTE may be different from the metamodel of the syntax used in the selected analysis tool. Thus, it is necessary to translate EML model into another syntax (Figure 8.8). This transformation could be done with MDWorkbench environment (see section 8.9).

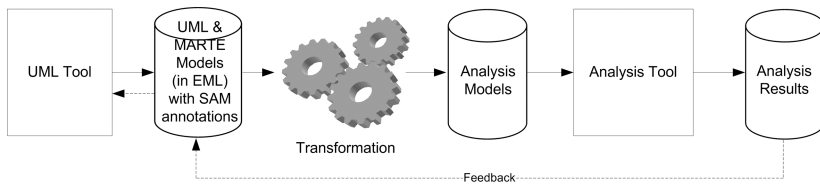


Figure 8.8. *Process of models analysis*

In addition, in this level, the PSM can be annotated with time constraints and elements of MARTE PAM sub-profile in order to analyze the system performances. The results of this analysis are still approximate. Precise performance analysis can be carried out in the Detailed Modeling Level.

8.8. Detailed modeling level

The DML defines the platform at a clock cycle tick precise definition, where the final target RTL model can be generated. At this level, hardware specification is finalized relying on generated hardware components or an existing IP block.

8.8.1. Platform model

The platform defines the structural hardware components, which is a refinement of the PM defined in the above level, EML. A component diagram is used to model it. MARTE HRM – *Hardware Resource Modeling* – sub-profile is used to define which kind of elements each object represents, such as ASIC – *Application Specific Integrated Circuit* –, PLD – *Programmable Logic Device* –, Clock, etc. MARTE SRM – *Software Resource Modeling* – sub-profile is used to model operating system properties, like tasks and virtual memory. MARTE SRM elements are not addressed here. All components of the platform must be stereotyped with MARTE HRM elements.

A platform is defined as a set of components connected through ports. For each port, a stereotype, which defines a communication protocol, is attached. A library is associated with each protocol stereotype, used in code generation. Figure 8.9 shows the elements in a platform model.

A component with a «HwClock» must be present in the platform that is used to allow performance analysis and synchronous component code generation.

Component diagram. The component diagram contains the platform resources. At least two stereotypes must be present for each component: «HwLogical» and «HwPhysical». Both must be present to characterize DML. Components are used to

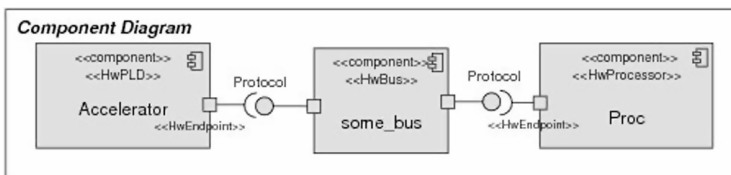


Figure 8.9. Platform model

model the platform as they are considered to be reusable units that offer services, abstracting their behavior. Each component must be identified by an IP number and version, which allows IP definition and reuse.

Components are connected together by UML ports, where the ports contain the stereotype «HwEndPoint». An endpoint is an interaction point to communicate with the component.

Protocol definition. Inter-component communication is done by communication protocols. Protocols are defined as interfaces, where buses ports offer a protocol and other components ports require it. The set of available protocols is platform-dependent and the code generation tool must be aware of the available protocols.

8.8.2. Allocation model

Allocation at this level involves defining where functional objects – MoC Components – are placed in platform ones – MARTE HRM stereotyped objects –. Functional components are allocated onto platform ones (Figure 8.10). Application components are logical units with behavior. Such behavior will be executed in/by a platform component. It is important to remark that a component whose behavior definition contains

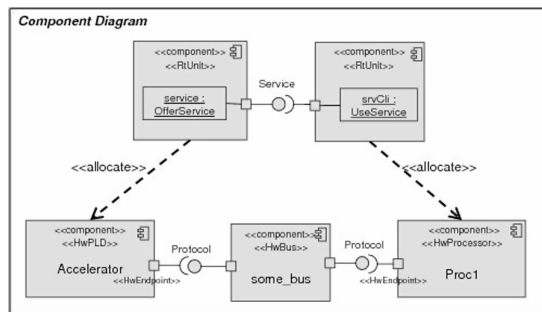


Figure 8.10. Allocation model

a state machine must be allocated to a component connected to a clock or to a processor.

8.9. Tooling Support

The MDD/MARTE design process presented above is defined to be as independent as possible from the implementation tools, so that it could be instantiated into any other tool (open-source modeler, java/EMF model transformer, etc.). In any case, for the MOPCOM project, this process relies on three main tools:

- Rhapsody [TEL], a UML Modeler targeted to RT – *Real Time* – embedded applications, to model the application and the platform,
- Kermeta, a Metamodeler [INR, MUL 05], to formalize (concepts and constraints), validate the metamodels and specify the transformation steps,
- MDWorkbench platform [SOD], a model-driven workbench to transform models (model-to-model) and to generate code or documentation from models (model-to-text).

Figure 8.11 depicts our tooling workflow; all tools being based on MDA standards from OMG (MDA, UML, MOF, XMI) and Eclipse (EMF, EMOF, ECore). For MOPCOM, an implementation of MARTE profile in Rhapsody has been developed.

8.9.1. *Process validation through metamodeling with Kermeta*

Kermeta environment from INRIA is devoted to model manipulations (composition, merge, etc.). It relies on an imperative object-oriented language and gives operational semantics to metamodels in a non invasive way, taking advantage of a built-in *aspect mechanism* to weave Kermeta

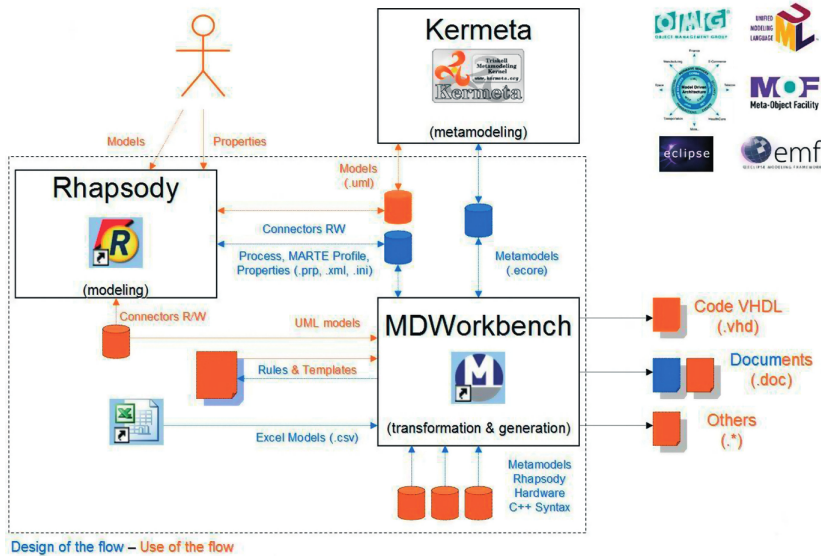


Figure 8.11. MOPCOM process tools interactions

code to Ecore model elements. For model validation, Kermeta provides the same capacities as OCL to define rules and check models. So, a Model Checker based on Kermeta environment has been developed. It was used to validate models transformations and to check models compliance to MOPCOM modeling rules at each step of the MOPCOM process (see Figure 8.1).

8.9.2. Model transformation and generation with MDWorkbench platform

The MDWorkbench platform from Sodius includes a complete environment to handle metamodels and models, and to design, execute, test and deploy model-to-model transformation rules and model-to-text generation rules. It is seamlessly integrated into Rhapsody, known as RulesPlayer and RulesComposer. The RulesPlayer can be

seen as a black-box runtime generation engine, while the RulesComposer is the rule editor, for designing and modifying the transformation and generation rule sets. MDWorkbench is delivered as an Eclipse plugin, and built as a model-driven extension to this powerful environment with many helpful capabilities (edition, windowing, debug, data handling, versioning, etc.). It includes the required mechanisms based on EMF/Ecore, to build, browse and import/export any metamodel, such as the Rhapsody metamodel and model.

The generator is delivered as a white-box Rhapsody add-on. All transformation and generation rules are available for customization with MQL – *Model Query Language* – dedicated to model transformation, and TGL – *Text Generation Language* – for code or doc generation. MQL and TGL offer java-like main constructs (declarations, selections and loops) and a high-level dotted notation to handle lists of model elements.

MDWorkbench incorporates a powerful document generator, based on a gateway with Microsoft Word®. An XML – *Extensible Markup Language* – document schema is provided that enables users to define their own document templates within Word, in compliance with their company's graphic policy or with any standard. This greatly enhances the power of a model-driven design approach where the application/platform models become the reference, and where the development documentation is automatically generated from the model.

8.10. HDL Code Generation

Code generation is a capacity generally supported by a MDD process. For ESL, the target language is an HDL such as VHDL or SystemC. A VHDL code generator for Rhapsody, presented hereafter, has been developed in the MOPCOM project.

8.10.1. *VHDL code generation*

VHDL code generator input is a DML model, lowest abstraction level within the process, which includes the application and platform packages, as well as the allocation of the application class instances on the platform class instances. As usual, package class/object and statechart diagrams feed code generation, which targets synthesizable VHDL code.

The structure part is derived from the platform model, where VHDL entities are derived from instances (and instance hierarchy) of platform classes. Obviously, a UML port is not at all mapped to a VHDL port, but corresponds to a communication channel between system blocks, which also represents the entity port set. Data and control VHDL ports are determined according to the UML port and its mapping on a model of computation (and communication protocol) and can be imported from existing libraries. Additional VHDL properties enable definition of a clock (with edge), and optional asynchronous/synchronous resets (with polarity).

The behavioral part is derived from the application model, where VHDL architectures are mainly issued from attributes, operations and state machines. All the required data types are defined into associated packages, according to the UML types (enumerations, language-defined, structure and hierarchical types, constants, etc.). The VHDL processes handle internal signals and variables that are declared according to data definitions in the scope of each class. Nearly all concepts of UML state machines are supported by the generator. Briefly, a finite state machine leads to the definition of an enumerated type for the active state, one per composite state (containing sub-states). The code structure is based on an edge-clocked case VHDL statement, and all trigger, guard and action expressions (on transitions, entering, in or exiting states) can be generated either in line, or in single procedures.

The allocation package brings additional information about the mapping of the application on the platform. The generator combines the declared entity ports and the data/control needs of the architecture to map the components and if required (if not point-to-point), instantiate the control code (or state machine) of the communication channel protocols. Depending on the communication channel, several basic mechanisms are provided to handle events (transient or registered) and the required logic is automatically inserted.

The generation process consists of two steps: the first step is a model transformation from Rhapsody to an intermediate hardware model; the second step is a generation from hardware model to VHDL code. The hardware model conforms with a hardware metamodel which gathers the main semantic concepts that are required to describe a complex electronic device at the Register Transfer Level.

8.10.2. *Rhapsody integration*

The HDL generator is currently embedded into Rhapsody in C++ and VHDL generation is just another environment of the Rhapsody configuration. Extra properties have been defined, in order to setup the generator, and to select the coding style, naming rules and generation parameters. VHDL code can be edited directly into the UML tool, as each in-the-scope class is associated with a specification file (VHDL package) and an implementation file (VHDL entity/architecture). The usual build-and-make phase has been converted into a possible call to a VHDL synthesizer, and a current bridge with web-free Xilinx XST. The bridge allows any warning and error messages to be displayed back into the Rhapsody build window, with dynamic link to the specified code line.

It is also possible implement interfaces with some other tools, either adjacent modeling tools such as Doors®(Telelogic) or Matlab/Simulink®(the Mathworks), or EDA – *Electronic*

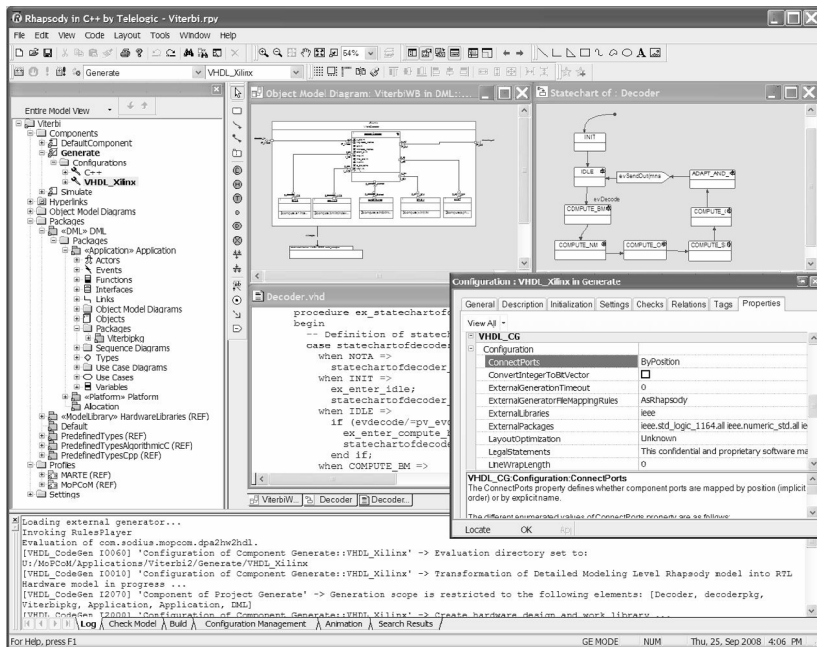


Figure 8.12. VHDL code generator integration into Rhapsody

Design Automation – downstream tools such as EDK®(Xilinx), Altera®(SoPC Builder) or any other modeling tools from EDA tool suppliers.

8.11. Conclusion

In this paper, we have discussed an ESL process based on MDD and MARTE profile. This process emphasizes application and platform modeling at different levels of abstraction and the allocation of the application models to the platform models. For each level, we presented the selected MARTE stereotypes and the constraints related to their use. We have also outlined the MDD tooling developed to support the process: for example a MARTE profile implementation in a UML modeler, and a VHDL code generator.

We believe that the emergence of the MARTE profile will make the use of MDD methodology widespread in the ESL domain. UML and MDD methodology are supported by a large number of commercial and freeware development tools that will offer new possibilities to the ESL community.

8.12. Acknowledgements

The UML/MDD approach presented above was developed in the RNTL research program MOPCOM SoC/SoPC supported by the French Agence Nationale de la Recherche (contract 2006 TLOG 022 01), the “Media and Networks” “cluster of clusters” and the Brittany and Pays de la Loire regions.

8.13. Bibliography

- [ART 08] ARTHURS G., White paper: Model-based system engineering, IBM, October 2008.
- [BOE 88] BOEHM B. W., “A Spiral Model of Software Development and Enhancement”, *Computer*, May 1988, p. 61-72.
- [BUC 02] BUCK J., HA S., LEE E. A., MESSERSCHMITT D. G., “Ptolemy: a framework for simulating and prototyping heterogeneous systems”, *IEEE*, vol. 10, 2002, p. 527–543, Kluwer Academic Publishers.
- [CAL 08] CALVEZ J., The MCSE Methodology overview, report 2008, CoFluent Design.
- [CEA] CEA, Papyrus UML2 Modeler, <http://www.papyrusuml.org>.
- [COF] COFLUENT_DESIGN, CoFluent Studio, <http://www.cofluentdesign.com/>.
- [GAJ 83] GAJSKI D. D., KUHN R. H., “New VLSI Tools”, *Computer*, vol. 16, num. 12, 1983, p. 11–14, IEEE Computer Society Press.
- [GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Num. ISBN 0-201-63361-2, Addison-Wesley, 1995.

- [GER 02] GERARD S., TERRIER F., TANGUY Y., “Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML”, SPRINGLINK, Ed., *Advances in Object-Oriented Information Systems*, vol. 2426/2002 of *Lecture Notes in Computer Science*, 2002, p. 260-269.
- [GER 03] GERARD S., TERRIER F., “UML for real-time: which native concepts to use?”, *ACM*, vol. 13, 2003, p. 17–51, Kluwer Academic Publishers.
- [HAC 07] HACHEMANI R., PALICOT J., MOY C., “A new standard recognition sensor for cognitive radio terminals”, *EURASIP*, Kessariani, Greece, 2007.
- [INR] INRIA, Kermeta metaprogramming environment, <http://www.kermeta.org>.
- [ITR 07] ITRS, Design, report 2007, International Technology Roadmap for Semiconductors.
- [KAO 04] KAOUANE L., AKIL M., GRANDPIERRE T., SOREL Y., “A methodology to implement real-time applications onto reconfigurable circuits”, *J. Supercomput.*, vol. 30, num. 3, 2004, p. 283–301, Kluwer Academic Publishers.
- [KAS 07] KASUYA A., TESFAYE T., “Verification methodologies in a TLM-to-RTL design flow”, *DAC '07: Proceedings of the 44th annual conference on Design automation*, New York, NY, USA, 2007, ACM, p. 199–204.
- [KOU 05] KOUDRI A., MEFTALI S., DEKEYSER J.-L., “IP integration in embedded systems modeling”, *14th IP Based SoC Design Conference (IP-SoC 2005)*, Grenoble, France, December 2005.
- [KOU 08] KOUDRI A., AL., “Using MARTE in a Co-Design Methodology”, *DATE*, 2008, Workshop MARTE.
- [LEB 08] LE BOLZER F., GUILLOUARD S., GUGUEN C., FONTAINE P., MONNIER R., “Prodim@ges - A new Video Production Environment based on IP wireless and optical links”, *NEM'SUMMIT*, Saint-Malo, France, October 2008.
- [MIT 01] MITOLA JOSEPH I., “Cognitive radio for flexible mobile multimedia communications”, *Mob. Netw. Appl.*, vol. 6, num. 5, 2001, p. 435–441, Kluwer Academic Publishers.

- [MOP 07] MoPCoM, MoPCoM SoC/SoPC Project, <http://www.mopcom.fr>, 2007.
- [MUL 05] MULLER P.-A., FLEUREY F., JÉZÉQUEL J.-M., “Weaving Executability into Object-Oriented Meta-Languages”, *Proc. of MODELS/UML*, LNCS, Montego Bay, Jamaica, 2005, Springer.
- [OMG 03] OMG, MDA Guide Version 1.0.1, report 2003, Object Management Group.
- [OMG 05a] OMG, UML Profile for Schedulability, Performance, and Time, version 1.1, report num. formal/2005-01-02, 2005, Object Management Group.
- [OMG 05b] OMG, A UML Profile for SoC, report num. Realtime - 2005-04-12, 2005.
- [OMG 06a] OMG, Object Constraint Language, report num. formal/2006-05-01, 2006, Object Management Group.
- [OMG 06b] OMG, UML 2.1 Infrastructure, report num. ptc/06-04-03, 2006, Object Management Group.
- [OMG 07] OMG, UML Profile for MARTE, Beta 1, report num. ptc/07-08-04, 2007, Object Management Group.
- [OMG 08a] OMG, Systems Modeling Language Specification v1.1, report num. ptc/2008-05-16, 2008, Object Management Group.
- [OMG 08b] OMG, UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms, report num. formal-2008-04-05, 2008, Object Management Group.
- [OSC 05] OSCI, IEEE Standard SystemC Language Reference Manual, report num. IEEE Std 1666-2005, 2005, IEEE Computer Society.
- [PIE 08] PIEL E., ATTITALAH R. B., MARQUET P., MEFTALI S., NIAR S., ETIEN A., DEKEYSER J.-L., BOULET P., *Gaspard2: from MARTE to SystemC Simulation*, 2008.
- [RIC 05] RICCOBENE E., SCANDURRA P., ROSTI A., BOCCHIO S., “A SoC Design Methodology Involving a UML 2.0 Profile for SystemC”, *Proc. of the conference on Design, Automation and Test in Europe*, Munich, Germany, March 2005, IEEE Computer Society, p. 704-709.

- [RIC 07] RICCOBENE E., SCANDURRA P., ROSTI A., BOCCHIO S., “Designing a Unified Process for Embedded Systems”, *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, Braga, Portugal, March 2007, IEEE Computer Society.
- [SAN 04] SANGIOVANNI-VINCENTELLI A., CARLONI L., BERNARDINIS F. D., SGROI M., “Benefits and challenges for platform-based design”, *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, New York, NY, USA, 2004, ACM, p. 409–414.
- [SIN 04] SINGHOFF F., LEGRAND J., NANA L., MARCÉ L., “Cheddar : a Flexible Real Time Scheduling Framework”, *Proc. of International Conference on Special Interest Group on Ada (SIGAda)*, Atlanta, Georgia, USA, November 2004, ACM.
- [SOD] SODIUS, MDWorkbench platform, <http://www.mdworkbench.com>.
- [TEL] TELELOGIC, Rhapsody UML modeler, <http://www.telelogic.com/products/rhapsody/index.cfm>.

List of Authors

Denis AULAGNIER
Thales Aerospace Division
Brest
France

Jean-Philippe BABAU
LISyC
UBO
Brest
France

Benoit BAUDRY
INRIA / IRISA
Rennes
France

Mireille BLAY-FORNARINO
Engineering School of Technology
University of Nice Sophia
Antipolis
France

Daniela CANCILA
CEA LIST
Model-Driven Engineering Labs (LISE)
Gif sur Yvette
France

Joël CHAMPEAU
ENSIETA
Brest
France

Philippe DHAUSSY
LISyC
ENSIETA
Brest
France

Huascar ESPINOZA
CEA LIST
Model-Driven Engineering Labs (LISE)
Gif sur Yvette
France

Christophe GASTON
CEA LIST
Model-Driven Engineering Labs
Gif sur Yvette
France

Sébastien GERARD
CEA LIST
Model-Driven Engineering Labs (LISE)
Gif sur Yvette
France

Ali KOUDRI
Thales Aerospace Division
Brest
France

Pascale LE GALL
University of Evry
LaMI
France

Stéphane LECOMTE
Thomson R&D France
Corporate Research – Networking Lab
Cesson-Sévigné
France

Pierre LERAY
Supélec
Cesson-Sévigné
France

Tom MENS
University of Mons
Belgium

Gilles PERROUIN
INRIA
Rennes
France

Dorina C. PETRIU
Carleton University
Department of Systems and Computer Engineering
Ottawa
Canada

Pierre-Yves PILLAIN
LISyC
ENSIETA
Brest
France

Chris RAISTRICK
Kennedy Carter Limited
East Clandon
Surrey
England

Nicolas RAPIN
CEA LIST
Model-Driven Engineering Labs
Gif sur Yvette
France

Sylvain ROBERT
CEA LIST
Saclay
France

Antonio SABETTA
ISTI-CNR
Pisa
Italy

Bran SELIC
Malina Software Corporation
Ontario
Canada

Philippe SOULARD
Sodius
Nantes
France

Assia TOUIL
University of Evry
LaMI
France

Jorgiano VIDAL
University of South Brittany
Lab-STICC
Lorient
France

Index

C, D

code generation, 22, 23, 31,
32, 34, 37, 39, 41
combinatorial testing, 52, 69
conformance testing, 73, 76,
86, 87
coverage criteria, 97, 100, 101
delay characteristic, 169
driver, 168-176, 181, 184-187,
190-194

E, F

embedded systems, 105
ESL 201-205, 225, 228, 229
exhaustive simulation, 168,
169, 175, 191, 194
FPGA, 201, 209

I, L

Input/Output Symbolic
Transition Systems, 80
LQN, 144-148, 155-161, 165

M

MARTE, 105-107, 110-136,
140, 149, 152-154, 165, 201,
202, 205, 206, 208-223, 228,
229
MDA, 22-24, 29, 34, 41, 202,
206, 223,
MDD, 201, 206, 210, 223,
225, 228
metamodel-based test
generation, 69
model transformation, 2-9,
12-14, 17, 19, 140
model-driven engineering, 1,
135
modeling languages, 106

P, R

performance analysis, 139,
151, 164, 165, 166
real-time, 168, 171, 196-200

S

SoC/SoPC, 201-203, 206,
210, 211, 229
survey, 1

symbolic execution, 73-101
SysML, 105-107, 109, 111-137

T

taxonomy, 4, 17
test adequacy criteria, 45, 49, 61, 62, 64
test input generation, 58

test purposes 77, 79, 88, 96-101

U

UML, 22, 24, 25, 29, 30-34, 38-42, 106-113, 115, 116, 118, 121, 125-131, 133, 135, 136, 140, 148, 151-156, 158, 159, 164-166, 202, 204-210, 213, 214, 220-223, 226, 227